

Using

ADOBE® LIVECYCLE® DATA SERVICES ES2

Version 3.1



© 2010 Adobe Systems Incorporated. All rights reserved.

Using Adobe® LiveCycle® Data Services ES2 version 3.1.

This user guide is protected under copyright law, furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

This user guide is licensed for use under the terms of the Creative Commons Attribution Non-Commercial 3.0 License. This License allows users to copy, distribute, and transmit the user guide for noncommercial purposes only so long as (1) proper attribution to Adobe is given as the owner of the user guide; and (2) any reuse or distribution of the user guide contains a notice that use of the user guide is governed by these terms. The best way to provide notice is to include the following link. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/>

Adobe, the Adobe logo, ActionScript, Adobe AIR, AIR, Dreamweaver, Flash, Flash Builder, Flex, JRun, and LiveCycle are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Apple, Macintosh, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries. IBM is a trademark of International Business Machines Corporation in the United States, other countries, or both. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Solaris is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark of The Open Group in the US and other countries. All other trademarks are the property of their respective owners.

Updated Information/Additional Third Party Code Information available at <http://www.adobe.com/go/thirdparty>.

Portions include software under the following terms:

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

This product includes software developed by the OpenSymphony Group (<http://www.opensymphony.com/>)

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.

This software is based in part on the work of the Independent JPEG Group.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Chapter 1: Getting started with LiveCycle Data Services

Introducing LiveCycle Data Services	1
Building and deploying LiveCycle Data Services applications	9

Chapter 2: System architecture

Client and server architecture	27
Channels and endpoints	38
Managing session data	73
Data serialization	79

Chapter 3: Controlling data traffic

Data throttling	100
Deserialization validators	104
Advanced data tuning	106
Message delivery with adaptive polling	116
Measuring message processing performance	121

Chapter 4: RPC services

Using RPC services	135
--------------------------	-----

Chapter 5: Message Service

Using the Message Service	190
Connecting to the Java Message Service (JMS)	214

Chapter 6: Data Management Service

Introducing the Data Management Service	221
Data Management Service clients	226
Data Management Service configuration	244
Custom assemblers	254
Standard assemblers	268
Hierarchical data	293
Data paging	303
Occasionally connected clients	310
Server push	323

Chapter 7: Model-driven applications

Building your first model-driven application	326
Building an offline-enabled application	333
Customizing client-side functionality	341
Customizing server-side functionality	348
Generating database tables from a model	364
Creating a client for an existing service destination	366
Configuring a data source	371
Configuring RDS on the server	371

Building the client application	372
Using server-side logging with the Model Assembler	383
Setting Hibernate properties for a model in a Hibernate configuration file	383
Configuring the model deployment service	383
Entity utility	384
 Chapter 8: Edge Server	
Connecting an Edge Server to a server in the application tier	391
Example application configuration	394
Creating a merged configuration for client compilation	398
Edge Server authentication and authorization	400
Restricting access from the Edge Server with white lists and black lists	401
Connecting Flex clients to an Edge Server	401
Handling missing Java types at the edge tier	403
JMX management	403
 Chapter 9: Generating PDF documents	
About the PDF generation feature	404
Using the PDF generation feature	404
 Chapter 10: Run-time configuration	
About run-time configuration	411
Configuring components with a bootstrap service	411
Configuring components with a remote object	414
Using assemblers with run-time configuration	416
Accessing dynamic components with a Flex client application	417
 Chapter 11: Administering LiveCycle Data Services applications	
Logging	420
Security	429
Clustering	442
Integrating Flex applications with portal servers	451
 Chapter 12: Additional programming topics	
The Ajax client library	460
Extending applications with factories	476

Chapter 1: Getting started with LiveCycle Data Services

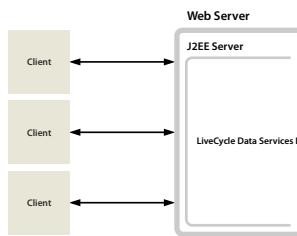
Introducing LiveCycle Data Services

Adobe® LiveCycle® Data Services provides highly scalable remote access, messaging, and data management services for use with client-side applications built in Adobe® Flex® or Adobe® AIR™.

LiveCycle Data Services overview

LiveCycle Data Services provides a set of services that lets you connect a client-side application to server-side data, and pass data among multiple clients connected to the server. LiveCycle Data Services synchronizes data sharing among clients, performs data push and data conflict management, and implements real-time messaging between clients.

A LiveCycle Data Services application consists of two parts: a client-side application and a server-side J2EE web application. The following figure shows this architecture:



The client-side application

A LiveCycle Data Services client application is typically an Flex or AIR application. Flex and AIR applications use Flex components to communicate with the LiveCycle Data Services server, including the RemoteObject, HTTPService, WebService, Producer, Consumer, and DataService components. The HTTPService, WebService, Producer, and Consumer components are part of the Flex Software Development Kit (SDK). To use the DataService component, configure your development environment to use the LiveCycle Data Services SWC files. For more information, see “[Building and deploying LiveCycle Data Services applications](#)” on page 9.

Although you typically use Flex or AIR to develop the client-side application, you can develop the client as a combination of Flex, HTML, and JavaScript. Or, you can develop it in HTML and JavaScript by using the Ajax client library to communicate with LiveCycle Data Services. For more information on using the Ajax client library, see “[The Ajax client library](#)” on page 460.

The LiveCycle Data Services server

The LiveCycle Data Services server consists of a J2EE web application and a highly scalable socket server running on a J2EE application server. The LiveCycle Data Services installer creates three web applications that you can use as the basis of your application development. For more information on using these web applications, see “[Building and deploying LiveCycle Data Services applications](#)” on page 9.

Configure an existing J2EE web application to support LiveCycle Data Services by performing the following steps:

- 1 Add the LiveCycle Data Services JAR files and dependent JAR files to the WEB-INF/lib directory.

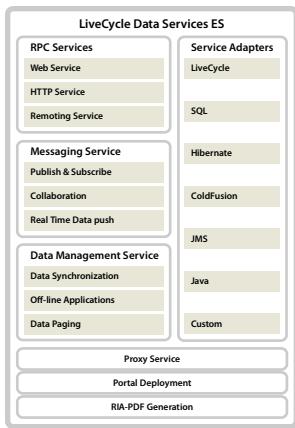
- 2 Edit the LiveCycle Data Services configuration files in the WEB-INF/flex directory.
- 3 Define MessageBrokerServlet and a session listener in WEB-INF/web.xml.

Versions of LiveCycle Data Services

You can download a free developer version of LiveCycle Data Services with certain restrictions on its use. For more information, see the LiveCycle Enterprise Suite page on www.adobe.com.

LiveCycle Data Services features

The following figure shows the main features of LiveCycle Data Services:



LiveCycle Data Services core features

The LiveCycle Data Services core features include the RPC services, Message Service, and Data Management Service.

RPC services

The Remote Procedure Call (RPC) services are designed for applications in which a call and response model is a good choice for accessing external data. RPC services let a client application make asynchronous requests to remote services that process the requests and then return data directly to the client. You can access data through client-side RPC components that include HTTP GET or POST (HTTP services), SOAP (web services), or Java objects (remote object services).

Use RPC components when you want to provide enterprise functionality, such as proxying of service traffic from different domains, client authentication, whitelists of permitted RPC service URLs, server-side logging, localization support, and centralized management of RPC services. LiveCycle Data Services lets you use RemoteObject components to access remote Java objects without configuring them as SOAP-compliant web services.

A client-side RPC component calls a remote service. The component then stores the response data from the service in an ActionScript object from which you can easily obtain the data. The client-side RPC components are the HTTPService, WebService, and RemoteObject components.

Note: You can use Flex SDK without the LiveCycle Data Services proxy service to call HTTP services or web services directly. You cannot use RemoteObject components without LiveCycle Data Services or Adobe® ColdFusion®.

For more information, see “[Using RPC services](#)” on page 135.

Message Service

The Message Service lets client applications communicate asynchronously by passing messages back and forth through the server. A message defines properties such as a unique identifier, LiveCycle Data Services headers, any custom headers, and a message body.

Client applications that send messages are called message *producers*. You define a producer in a Flex application by using the Producer component. Client applications that receive messages are called message *consumers*. You define a consumer in a Flex application by using the Consumer component. A Consumer component subscribes to a server-side destination and receives messages that a Producer component sends to that destination. For more information on messaging, see “[Using the Message Service](#)” on page 190.

The Message Service also supports bridging to JMS topics and queues on an embedded or external JMS server by using the JMSAdapter. Bridging lets Flex client applications exchange messages with Java client applications. For more information, see “[Connecting to the Java Message Service \(JMS\)](#)” on page 214.

Data Management Service

The Data Management Service lets you create applications that work with distributed data. By using the Data Management Service, you build applications that provide real-time data synchronization, data replication, on-demand data paging, and occasionally connected application services. You can manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships. You can also use data adapters to integrate with data resources, such as a database.

Note: The Data Management Service is not available in BlazeDS.

A client-side DataService component calls methods on a server-side Data Management Service destination. Use this component to perform activities such as filling client-side data collections with data from remote data sources and synchronizing the client and server versions of data. Changes made to the data at the client side are tracked automatically using property change events.

When the user is ready to submit their changes, the changes are sent to a service running on the server. This service then passes the changes to a server-side adapter, which checks for conflicts and commits the changes. The adapter can be an interface you write, or one of the supplied adapters that work with a standard persistence layer such as SQL or Hibernate. After the changes are committed, the Data Management Service pushes these changes to any other clients looking at the same data.

For more information, see “[Introducing the Data Management Service](#)” on page 221.

Service adapters

LiveCycle Data Services lets you access many different persistent data stores and databases including Hibernate, SQL, JMS, and other data persistence mechanisms. A Service Adapter is responsible for updating the persistent data store on the server in a manner appropriate to the specific data store type. The adapter architecture is customizable to let you integrate with any type of messaging or back-end persistence system.

The message-based framework

LiveCycle Data Services uses a message-based framework to send data back and forth between the client and server. LiveCycle Data Services uses two primary exchange patterns between server and client. In the first pattern, the request-response pattern, the client sends a request to the server to be processed. The server returns a response to the client containing the processing outcome. The RPC services use this pattern.

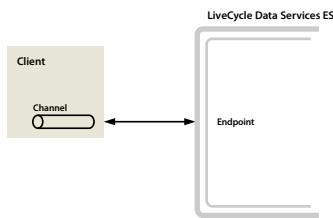
The second pattern is the publish-subscribe pattern where the server routes published messages to the set of clients that have subscribed to receive them. The Message Service and Data Management Service use this pattern to push data to interested clients. The Message Service and Data Management Service also use the request-response pattern to issue commands, publish messages, and interact with data on the server.

Channels and endpoints

To send messages across the network, the client uses channels. A channel encapsulates message formats, network protocols, and network behaviors to decouple them from services, destinations, and application code. A channel formats and translates messages into a network-specific form and delivers them to an endpoint on the server.

Channels also impose an order to the flow of messages sent to the server and the order of corresponding responses. Order is important to ensure that interactions between the client and server occur in a consistent, predictable fashion.

Channels communicate with Java-based endpoints on the server. An endpoint unmarshals messages in a protocol-specific manner and then passes the messages in generic Java form to the message broker. The message broker determines where to send messages, and routes them to the appropriate service destination.



For more information on channels and endpoints, see “[Client and server architecture](#)” on page 27.

Channel types

LiveCycle Data Services includes several types of channels, including standard and secure Real Time Messaging Protocol (RTMP) channels and channels that support binary Action Message Format (AMF) and its text-based XML representation called AMFX. AMF and HTTP channels support non-polling request-response patterns and client polling patterns to simulate real-time messaging. The RTMP channels and streaming AMF and HTTP channels provide true data streaming for real-time messaging.

LiveCycle Data Services summary of features

The following table summarizes some of the main features of LiveCycle Data Services:

Feature	Description
Client-server synchronization	Automatic and manual synchronization of a common set of data on multiple clients and server-side data resources. Also supports offline client-side data persistence for occasionally connected clients. Removes the complexity and potential for error by providing a robust, high-performance data synchronization engine between client and server. It also can easily integrate with existing persistence solutions to provide an end-to-end solution.
Collaboration	Enables a client application to concurrently share data with other clients or servers. This model enables new application concepts like "co-browsing" and synchronous collaboration, which allow users to share experiences and work together in real time.
Data paging	Facilitates the paging of large data sets, enabling developers to focus on core application business logic instead of worrying about basic data management infrastructure.

Feature	Description
Data push	Enables data to automatically be pushed to the client application without polling. This highly scalable capability can push data to thousands of concurrent users to provide up-to-the-second views of critical data. Examples include stock trader applications, live resource monitoring, shop floor automation, and more.
Data traffic control	Provides a set of features for managing data traffic, such as data throttling, deserialization validation, reliable messaging, message prioritization, message filtering, and measuring message processing performance.
Model-driven development	Use Adobe application modeling technology to facilitate the development of data-centric applications.
Occasionally connected client	Handles temporary disconnects, ensuring reliable delivery of data to and from the client application. Provides support for the development of offline and occasionally connected applications that run in the browser or on the desktop. LiveCycle Data Services takes advantage of the scalable local SQLite database in AIR to store data, synchronize it back to the server, and rationalize any changes or conflicts.
Portal service integration	Configure a Flex client applications as local portlets hosted on JBoss Portal, Oracle WebLogic Portal, or IBM WebSphere Portal.
Proxy service	Enables communication between clients and domains that they cannot access directly, due to security restrictions, allowing you to integrate multiple services with a single application. By using the Proxy Service, you do not have to configure a separate web application to work with web services or HTTP services.
Publish and subscribe messaging	Provides a messaging infrastructure that integrates with existing messaging systems such as JMS. This service enables messages to be exchanged in real time between browser clients and the server. It allows Flex clients to publish and subscribe to message topics with the same reliability, scalability, and overall quality of service as traditional thick client applications.
RIA-to-PDF generation	Users can generate template-driven PDF documents that include graphical assets from Flex applications, such as graphs and charts. The generated PDF documents can be orchestrated with other LiveCycle services and policy-protected to ensure only authorized access.
Software clustering	Handles failover when using stateful services and non-HTTP channels, such as RTMP, to ensure that Flex applications continue running in the event of server failure. The more common form of clustering using load balancers, usually in the form of hardware, is supported without any feature implementation.

Example LiveCycle Data Services applications

The following example applications show client-side and server-side code that you can compile and deploy to get started with LiveCycle Data Services. You typically use the following steps to build an application:

- 1 Configure a destination in the LiveCycle Data Services server used by the client application to communicate with the server. A destination is the server-side code that you connect to from the client. Configure a destination in one of the configuration files in the WEB-INF/flex directory of your web application.
- 2 Configure a channel used by the destination to send messages across the network. The channel encapsulates message formats, network protocols, and network behaviors and decouples them from services, destinations, and application code. Configure a channel in one of the configuration files in the WEB-INF/flex directory of your web application.
- 3 Write the Flex client application in MXML or ActionScript.
- 4 Compile the client application into a SWF file by using Adobe® Flash® Builder™ or the mxmcl compiler.
- 5 Deploy the SWF file to your LiveCycle Data Services web application.

Running the examples

The LiveCycle Data Services installer creates a directory structure on your computer that contains all of the resources necessary to build applications. As part of the installation, the installer creates three web applications that you can use as the basis of your development environment. The lcds-samples web application contains many LiveCycle Data Services examples.

You can run the following examples if you compile them for the lcds-samples web application and deploy them to the lcds-samples directory structure. For more information on building and running the examples, see “[Building and deploying LiveCycle Data Services applications](#)” on page 9.

RPC service example

The Remoting Service is one of the RPC services included with LiveCycle Data Services. The Remoting Service lets clients access methods of Plain Old Java Objects (POJOs) on the server.

In this example, you deploy a Java class, EchoService.java, on the server that echoes back a String passed to it from the client. The following code shows the definition of EchoService.java:

```
package remoting;
public class EchoService
{
    public String echo(String text) {
        return "Server says: I received '" + text + "' from you";
    }
}
```

The echo() method takes a String argument and returns it with additional text. After compiling EchoService.java, place EchoService.class in the WEB-INF/classes/remoting directory. Notice that the Java class does not have to import or reference any LiveCycle Data Services resources.

Define a destination, and reference one or more channels that transport the data. Configure EchoService.class as a remoting destination by editing the WEB-INF/flex/remoting-config.xml file and adding the following code:

```
<destination id="echoServiceDestination" channels="my-amf">
    <properties>
        <source>remoting.EchoService</source>
    </properties>
</destination>
```

The source element references the Java class, and the channels attribute references a channel called my-amf.

Define the my-amf channel in WEB-INF/flex/services-config.xml, as the following example shows:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/amf"
              class="flex.messaging.endpoints.AMFEEndpoint"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>
```

The channel definition specifies that the Flex client uses a non-polling AMFChannel to communicate with the AMFEndpoint on the server. Restart the LiveCycle Data Services server after making this change.

Note: If you deploy this application on the lcds-samples web application installed with LiveCycle Data Services, services-config.xml already contains a definition for the my-amf channel.

The Flex client application uses the RemoteObject component to access EchoService. The RemoteObject component uses the destination property to specify the destination. The user clicks the Button control to invoke the remote echo() method:

```
<?xml version="1.0"?>
<!!-- intro\intro_remoting.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%">

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            // Send the message in response to a Button click.
            private function echo():void {
                var text:String = ti.text;
                remoteObject.echo(text);
            }

            // Handle the received message.
            private function resultHandler(event:ResultEvent):void {
                ta.text += "Server responded: " + event.result + "\n";
            }

            // Handle a message fault.
            private function faultHandler(event:FaultEvent):void {
                ta.text += "Received fault: " + event.fault + "\n";
            }
        ]]>
    </mx:Script>

    <mx:RemoteObject id="remoteObject"
        destination="echoServiceDestination"
        result="resultHandler(event);"
        fault="faultHandler(event);"/>

    <mx:Label text="Enter a text for the server to echo"/>
    <mx:TextInput id="ti" text="Hello World!" />
    <mx:Button label="Send" click="echo();"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

Compile the client application into a SWF file by using Flash Builder or the mxmlc compiler, and then deploy it to your web application.

Message Service example

The Message Service lets client applications send and receive messages from other clients. In this example, create a Flex application that sends and receives messages from the same LiveCycle Data Services destination.

Define the messaging destination in WEB-INF/flex/messaging-config.xml, as the following example shows:

```
<destination id="MessagingDestination" channels="my-amf-poll"/>
```

Define the my-amf-poll channel in WEB-INF/flex/services-config.xml, as the following example shows:

```
<channel-definition id="my-amf-poll" class="mx.messaging.channels.AMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/messagebroker/amfpoll"
        class="flex.messaging.endpoints.AMFEEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-seconds>1</polling-interval-seconds>
    </properties>
</channel-definition>
```

This channel definition creates a polling channel with a polling interval of 1 second. Therefore, the client sends a poll message to the server every second to request new messages. Use a polling channel because it is the easiest way for the client to receive updates. Other options include polling with piggybacking, long-polling, and streaming. Restart the LiveCycle Data Services server after making this change.

The following Flex client application uses the Producer component to send a message to the destination, and the Consumer component to receive messages sent to the destination. To send the message, the Producer first creates an instance of the `AsyncMessage` class and then sets its `body` property to the message. Then, it calls the `Producer.send()` method to send it. To receive messages, the Consumer first calls the `Consumer.subscribe()` method to subscribe to messages sent to a specific destination.

```
<?xml version="1.0"?>
<!!-- intro\intro_messaging.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%"
    creationComplete="consumer.subscribe();">

    <mx:Script>
        <! [CDATA[
import mx.messaging.channels.AMFChannel;

import mx.messaging.ChannelSet;
import mx.messaging.events.MessageFaultEvent;
import mx.messaging.events.MessageEvent;
import mx.messaging.messages.AsyncMessage;
import mx.messaging.Producer;
import mx.messaging.Consumer;

// Send the message in response to a Button click.
private function sendMessage():void {
    var msg:AsyncMessage = new AsyncMessage();
    msg.body = "Foo";
    producer.send(msg);
}

// Handle the received message.
private function messageHandler(event:MessageEvent):void {
    ta.text += "Consumer received message: " + event.message.body + "\n";
}
```

```
// Handle a message fault.  
private function faultHandler(event:MessageFaultEvent):void {  
    ta.text += "Received fault: " + event.faultString + "\n";  
}  
]]>  
</mx:Script>  
  
<mx:Producer id="producer"  
destination="MessagingDestination"  
fault="faultHandler(event);"/>  
  
<mx:Consumer id="consumer"  
destination="MessagingDestination"  
fault="faultHandler(event);"  
message="messageHandler(event);"/>  
  
<mx:Button label="Send" click="sendMessage();"/>  
<mx:TextArea id="ta" width="100%" height="100%"/>  
</mx:Application>
```

Compile the client application into a SWF file by using Flash Builder or the mxmlc compiler, and then deploy it to your web application.

Building and deploying LiveCycle Data Services applications

Adobe LiveCycle Data Services applications consist of client-side code and server-side code. Client-side code is typically built with Flex in MXML and ActionScript and deployed as a SWF file. Server-side code is written in Java and deployed as Java class files or Java Archive (JAR) files. Every LiveCycle Data Services application has client-side code; however, you can implement an entire application without writing any server-side code.

For more information on the general application and deployment process for Flex applications, see the Flex documentation.

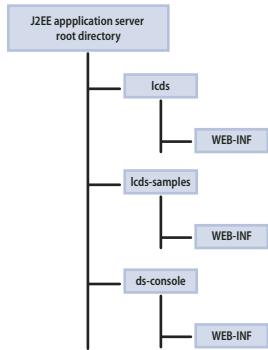
Setting up your development environment

LiveCycle Data Services applications consist of two parts: client-side code and server-side code. Before you start developing your application, configure your development environment, including the directory structure for your client-side source code and for your server-side source code.

Installation directory structure

The LiveCycle Data Services installer creates a directory structure on your computer that contains all of the resources necessary to build your application. As part of the installation, the installer creates three web applications that you can use as the basis of your development environment.

The following example shows the directory structure of the web applications installed with LiveCycle Data Services:



The installer gives you the option of installing the integrated Tomcat application server to host these web applications. Alternatively, you can install LiveCycle Data Services without installing Tomcat. Instead, you deploy the LiveCycle Data Services web application on your J2EE application server or servlet container.

The following table describes the directory structure of each web application:

Directory	Description
/lcds	The root directory of a web application. Contains the WEB-INF directory.
/lcds-samples	This directory also includes all files that must be accessible by the user's web browser, such as SWF files, JSP pages, HTML pages, Cascading Style Sheets, images, and JavaScript files. You can place these files directly in the web application root directory or in arbitrary subdirectories that do not use the reserved name WEB-INF.
/META-INF	Contains package and extension configuration data.
/WEB-INF	Contains the standard web application deployment descriptor (web.xml) that configures the LiveCycle Data Services web application. This directory can also contain a vendor-specific web application deployment descriptor.
/WEB-INF/classes	Contains Java class files and configuration files.
/WEB-INF/flex	Contains LiveCycle Data Services configuration files.
/WEB-INF/flex/libs	Contains SWC library files used when compiling a LiveCycle Data Services application.
/WEB-INF/flex/locale	Contains localization resource files used when compiling a LiveCycle Data Services application.
/WEB-INF/lib	Contains LiveCycle Data Services JAR files.
/WEB-INF/src	(Optional) Contains Java source code used by the web application.

Accessing a web application

To access a web application and the services provided by LiveCycle Data Services, you need the URL and port number associated with the web application. The following table describes how to access each web application assuming that you install LiveCycle Data Services with the integrated Tomcat application server.

Note: If you install LiveCycle Data Services into the directory structure of your J2EE application server or servlet container, modify the context root URL based on your development environment.

Application	Context root URL for Tomcat	Description
Sample application	http://localhost:8400/lcds-samples/	A sample web application that includes many LiveCycle Data Services examples. To start building your own applications, start by editing these samples.
Template application	http://localhost:8400/lcds/	A fully configured LiveCycle Data Services web application that contains no application code. You can use this application as a template to create your own web application.
Console application	http://localhost:8400/ds-console/	A console application that lets you view information about LiveCycle Data Services web applications.

If you install LiveCycle Data Services with the integrated Tomcat application server, you can also access the ROOT web application by using the following URL: <http://localhost:8400/>.

Creating a web application

To get started writing LiveCycle Data Services applications, you can edit the samples in the lcds-samples web application, add your application code to the lcds-samples web application, or add your application code to the empty lcds web application.

However, Adobe recommends leaving the lcds web application alone, and instead copying its contents to a new web application. That leaves the lcds web application empty so that you can use it as the template for creating web applications.

If you base a new web application on the lcds web application, make sure you change the port numbers of the RTMP and NIO channel definitions in the services-config.xml file of the new web application. Otherwise, the ports in the new web application will conflict with those in the lcds web application. Make sure the new port numbers you use aren't already used in other web applications, such as the lcds-samples web application.

Defining the directory structure for client-side code

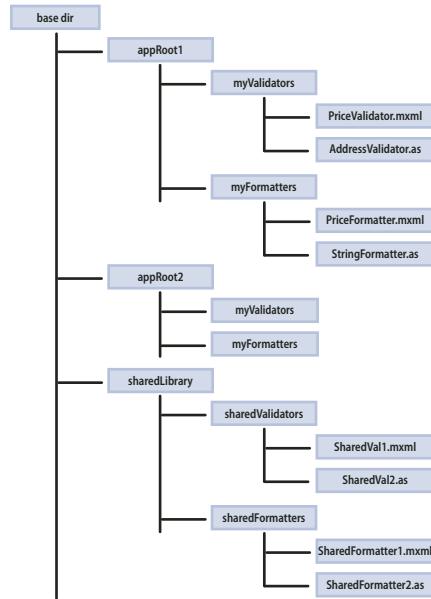
You develop LiveCycle Data Services client-side applications, and compile them in the same way that you compile applications that use the Flex Software Development Kit (SDK). That means you can use the compiler built in to Flash Builder, or the command line compiler, mxmlc, supplied with the Flex SDK.

When you develop applications, you have two choices for how you arrange the directory structure of your application:

- Define a directory structure on your computer outside any LiveCycle Data Services web application. Compile the application into a SWF file, and then deploy it, along with any run-time assets, to a LiveCycle Data Services web application.
- Define a directory structure in a LiveCycle Data Services web application. In this scenario, all of your source code and assets are stored in the web application. When you deploy the application, make sure to deploy only the application SWF file and run-time assets. Otherwise, you run the risk of deploying your source code on a production server.

You define each application in its own directory structure, with the local assets for the application under the root directory. For assets shared across applications, such as image files, you can define a directory that is accessible by all applications.

The following example shows two applications, appRoot1 and appRoot2. Each application has a subdirectory for local MXML and ActionScript components, and can also reference a library of shared components:



Defining the directory structure for server-side code

You develop the server-side part of a LiveCycle Data Services application in Java. For example, the client-side `RemoteObject` component lets you access the methods of server-side Java objects to return data to the client.

You also write Java classes to extend the functionality of the LiveCycle Data Services server. For example, a Data Management Service destination references one or more message channels that transport messages, and contains network- and server-related properties. The destination can also reference a *data adapter*, server-side code that lets the destination work with data through a particular type of interface such as a Java object. An *assembler class* is a Java class that interacts indirectly or directly with a data resource. For more information on assemblers, see “[Standard assemblers](#)” on page 268.

When you develop server-side code, you have several choices for how you arrange the directory structure of your application:

- Define a directory structure that corresponds to the package hierarchy of your Java source code outside any LiveCycle Data Services web application. Compile the Java code, and then deploy the corresponding class files and JAR files, along with any run-time assets, to a LiveCycle Data Services web application.
- Define a directory structure in a LiveCycle Data Services web application. In this scenario, all of your source code and assets are stored in the web application. When you deploy the application, make sure to deploy only the class and JAR files. Otherwise, you risk deploying source code on a production server.

The WEB-INF/classes and WEB-INF/lib directories are automatically included in the classpath of the web application. When you deploy your server-side code, place the compiled Java class files in the WEB-INF/classes directory. Place JAR files in the WEB-INF/lib directory.

Running the LiveCycle Data Services sample applications

When you install LiveCycle Data Services, the installer creates the lcds-samples web application that contains sample applications, including the 30 Minute Test Drive application. The sample applications demonstrate basic capabilities and best practices for developing LiveCycle Data Services applications.

The samples use an HSQLDB database that is installed in the *install_root*/sampledb directory. You must start the LiveCycle Data Services server and the samples database before you can run the LiveCycle Data Services samples. After starting the server and database, access the main sample application page by opening the following URL in a browser:

<http://localhost:8400/lcds-samples/>

The objective of the 30 Minute Test Drive is to give you, in a very short time, an understanding of how the LiveCycle Data Services works. Access the 30 Minute Test Drive application by opening the following URL in a browser:

<http://localhost:8400/lcds-samples/testdrive.htm>

The client-side source code for the samples is shipped in the lcds-samples/WEB-INF/flex-src/flex-src.zip file. To modify the client-side code, extract the flex-src.zip file into the lcds-samples directory, and then edit, compile, and deploy the modified examples. Editing the samples makes it easier to get started developing applications because you only have to modify existing code, rather than creating it from scratch.

Extract the client-side source code

- 1 Open lcds-samples/WEB-INF/flex-src/flex-src.zip file.
- 2 Extract the ZIP file into the lcds-samples directory.

Expanding the ZIP file adds a src directory to each sample in the lcds-samples directory. For example, the source code for the chat example, Chat.mxml, is written to the directory lcds-samples/testdrive-chat/src.

The server-side source code for these examples is shipped in the lcds-samples/WEB-INF/src/flex/samples directory. These source files are not zipped, but shipped in an expanded directory structure. To modify the server-side code you can edit and compile it in that directory structure, and then copy it to the lcds-samples directory to deploy it.

Run the sample applications

- 1 Change directory to *install_root*/sampledb.
- 2 Start the samples database by using the following command:

`startdb`

You can stop the database by using the command:

`stopdb`

- 3 Start LiveCycle Data Services.

How you start LiveCycle Data Services depends on your system.

- 4 Open the following URL in a browser:

<http://localhost:8400/lcds-samples/>

Building your client-side application

You write the client-side part of a LiveCycle Data Services application in Flex, and then use Flash Builder or the mxmclc command line compiler to compile it.

Before you begin

Before you begin to develop your client-side code, determine the files required to perform the compilation. Ensure that you configured your Flex installation to compile SWF files for LiveCycle Data Services applications.

Add the LiveCycle Data Services SWC files to the Flex SDK

To compile an application, Flash Builder and mxmclc reference the SWC library files that ship with the Flex SDK in subdirectories of the frameworks directory. In Flash Builder, the frameworks directories for different Flex SDK versions are in the *install_root/sdks/sdkversion* directory.

LiveCycle Data Services ships the following additional sets of SWC files. You must reference the set of SWC files that matches the Flex SDK you compile against.

- Flex 4 compatible SWC files in subdirectories of the *install_root/resources/lcds_swcs/FlexSDK4/frameworks* directory.
- Flex 3 compatible SWC files in subdirectories of the *install_root/resources/lcds_swcs/FlexSDK3/frameworks* directory.

Copy the files in the LiveCycle Data Services frameworks directory (including its subdirectories) to the corresponding Flex SDK frameworks directory.

Note: *Flex 3 compatible SWC files are also available in the WEB-INF/flex/libs directory of a LiveCycle Data Services web application. By default, Flash Builder adds these files to the library-path of a project that uses LiveCycle Data Services. However, the Flex 4 compatible versions of these files are not available in that location. You must add them manually from the install_root/resources/lcds_swcs/FlexSDK4/frameworks directory.*

LiveCycle Data Services provides the following SWC files:

- fds.swc and fiber.swc

The SWC library files that define LiveCycle Data Services. These SWC files must be included in the library path of the compiler.

- airfds.swc and playerfds.swc

The SWC files required to build LiveCycle Data Services applications for Flash Player (playerfds.swc) or AIR (airfds.swc). One of these SWC files must be included in the library path of the compiler.

For the default Flex SDK installation, playerfds.swc must be in the libs/player directory, and airfds.swc must be in the libs/air directory. The airfds.swc and playerfds.swc files must not both be available at the time of compilation. When you compile your application in Flash Builder, it automatically references the correct SWC file based on your project settings.

When you compile an application using mxmclc, by default the compiler references the flex-config.xml configuration file, which specifies to include the libs/player directory in the library path for Flash Player. When you compile an application for AIR, use the `load-config` option to the mxmclc compiler to specify the air-config.xml file, which specifies to include the libs/air directory in the library path.

- fds_rb.swc and fiber_rb.swc

The localized SWC files for LiveCycle Data Services. These SWC files must be in the library path of the compilation.

Specifying the services-config.xml file in a compilation

When you compile your Flex application, you typically specify the services-config.xml configuration file to the compiler. This file defines the channel URLs that the client-side Flex application uses to communicate with the LiveCycle Data Services server. Then the channel URLs are compiled into the resultant SWF file.

Both client-side and server-side code use the services-config.xml configuration file. If you change anything in services-config.xml, you usually have to recompile your client-side applications and restart your server-side application for the changes to take effect.

In Flash Builder, the appropriate services-config.xml file is included automatically based on the LiveCycle Data Services web application that you specified in the configuration of your Flash Builder project. When you use the mxmcl compiler, use the `services` option to specify the location of the file.

Note: You can also create channel definitions on the client in ActionScript or MXML. In that case, you might be able to omit the reference to the services-config.xml configuration file from the compiler. For more information, see “[About channels and endpoints](#)” on page 38.

Specifying the context root in a compilation

The services-config.xml configuration file typically uses the `context.root` token to specify the context root of a web application. At compile time, you use the compiler `context-root` option to specify that information.

During a compilation, Flash Builder automatically sets the value of the `context.root` token based on the LiveCycle Data Services web application that you specified in the configuration of your project. When you use the mxmcl compiler, use the `context-root` option to set it.

Using Flash Builder to compile client-side code

Flash Builder is an integrated development environment (IDE) for developing applications that use the Flex framework, MXML, Adobe Flash Player, AIR, ActionScript, LiveCycle Data Services, and the Flex charting components.

Flash Builder is built on top of Eclipse, an open-source IDE. It runs on Microsoft Windows, Apple Mac OS X, and Linux, and is available in several versions. Installation configuration options let you install Flash Builder as a plug-in to an existing Eclipse workbench installation, or to install it as a stand-alone application.

Using the stand-alone or plug-in configuration of Flash Builder

The Flash Builder installer provides the following two configuration options:

Plug-in configuration This configuration is for users who already use the Eclipse workbench, who already develop in Java, or who want to add the Flash Builder plug-ins to their toolkit of Eclipse plug-ins. Because Eclipse is an open, extensible platform, hundreds of plug-ins are available for many different development purposes. When you use the plug-in configuration, you can create a combined Java and Flex Eclipse project; for more information, see “[Creating a combined Java and Flex project in Eclipse](#)” on page 22.

Stand-alone configuration This configuration is a customized packaging of Eclipse and the Flash Builder plug-in created specifically for developing Flex and ActionScript applications. The stand-alone configuration is ideal for new users and users who intend to develop only Flex and ActionScript applications.

Both configurations provide the same functionality. You select the configuration when you install Flash Builder.

Most LiveCycle Data Services developers choose to use the Eclipse plug-in configuration. Then they develop the Java code that runs on the server in the same IDE that they use to develop the MXML and ActionScript code for the client Flex application.

Note: The stand-alone configuration of Flash Builder does not contain tools to edit Java code, however, you can install them. Select `Help > Software Updates > Find and Install` menu command to open the `Install/Update` dialog box. Then select `Search For New Features To Install`. In the results, select `Europa Discovery Site`, and then select the `Java Development package` to install.

If you aren’t sure which configuration to use, follow these guidelines:

- If you already use and have Eclipse 3.11 (or later) installed, select the plug-in configuration. On Macintosh, Eclipse 3.2 is the earliest version.

- If you don't have Eclipse installed and your primary focus is on developing Flex and ActionScript applications, select the stand-alone configuration. This configuration also lets you install other Eclipse plug-ins, so you can expand the scope of your development work in the future.

Create a Flash Builder project

Use this procedure to create a Flash Builder project to edit one of the samples shipped with the Test Drive application. The procedure for creating and configuring a new project is almost the same as the following procedure.

Note: When you use the plug-in configuration of Flash Builder, you can create a combined Java and Flex Eclipse project; for more information, see “[Creating a combined Java and Flex project in Eclipse](#)” on page 22.

For more information on the Test Drive application, see “[Running the LiveCycle Data Services sample applications](#)” on page 12.

- 1 Start Flash Builder.
- 2 Select File > New > Flex Project.
- 3 Enter a project name. You are editing an existing application, so use the exact name of the sample folder: **testdrive-chat**.
Note: If you are creating an empty project, you can name it anything that you want.
- 4 If you unzipped flex-src.zip in the lcds-samples directory, deselect the Use Default Location option, and specify the directory as *install_root/tomcat/webapps/lcds-samples/testdrive-chat*, or wherever you unzipped the file on your computer.

Note: By default, Flash Builder creates the project directory based on the project name and operating system. For example, if you are using the plug-in configuration of Flash Builder on Microsoft Windows, the default project directory is C:/Documents and Settings/USER_NAME/workspace/PROJECT_NAME.

- 5 Select the application type as Web (runs in Adobe® Flash® Player) to configure the application to run in the browser as a Flash Player application.

If you are creating an AIR application, select Desktop (runs In Adobe AIR). However, make sure that you do not have any server tokens in URLs in the configuration files. In the web application that ships with LiveCycle Data Services, server tokens are used in the channel endpoint URLs in the WEB-INF/flex/services-config.xml file, as the following example shows:

```
<endpoint
url="https://{{server.name}}:{{server.port}}/{{context.root}}/messagebroker/streamingamf"
class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
```

You would change that line to the following:

```
<endpoint url="http://your_server_name:8400/lcds/messagebroker/streamingamf"
class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
```

- 6 Select J2EE as the Application server type.
- 7 Select Use Remote Object Access.
- 8 Select LiveCycle Data Services.
- 9 Click Next.
- 10 Deselect Use Default Location For Local LiveCycle Data Services Server.
- 11 Set the root folder, root URL, and context root of your web application.

The root folder specifies the top-level directory of the web application (the directory that contains the WEB-INF directory). The root URL specifies the URL of the web application, and the context root specifies the root of the web application.

If you are using the integrated Tomcat application server, set the properties as follows:

Root folder: C:\lcds\tomcat\webapps\lcds-samples\

Root URL: http://localhost:8400/lcds-samples/

Context root: /lcds-samples/

Modify these settings as appropriate if you are not using the Tomcat application server.

12 Make sure that your LiveCycle Data Services server is running, and click Validate Configuration to ensure that your project is valid.

13 Clear the Output Folder field to set the directory of the compiled SWF file to the main project directory.

By default, Flash Builder writes the compiled SWF file to the bin-debug directory under the main project directory. To use a different output directory, specify it in the Output Folder field.

14 Click Next.

15 Set the name of the main application file to **Chat.mxml**, and click Finish.

Edit, compile, and deploy a LiveCycle Data Services application in Flash Builder

1 Open src/Chat.mxml in your Flash Builder project.

2 Edit Chat.mxml to change the definition of the TextArea control so that it displays an initial text string when the application starts:

```
<mx:TextArea id="log" width="100%" height="100%" text="My edited file!"/>
```

3 Save the file.

When you save the file, Flash Builder automatically compiles it. By default, the resultant SWF file is written to the C:/lcds/tomcat/webapps/lcds-samples/testdrive-chat/bin-debug directory, or the location you set for the Output directory for the project. You should have set the Output directory to the main project directory in the previous procedure.

Note: If you write the Chat.SWF file to any directory other than lcds-samples\testdrive-chat, deploy the SWF file by copying it to the lcds-samples\testdrive-chat directory.

4 Make sure that you have started the samples database and LiveCycle Data Services, as described in “[Running the LiveCycle Data Services sample applications](#)” on page 12.

5 Select Run > Run Chat to run the application.

You can also request the application in a browser by using the URL <http://localhost:8400/lcds-samples/testdrive-chat/index.html>.

Note: By default, Flash Builder creates a SWF file that contains debug information. When you are ready to deploy your final application, meaning one that does not contain debug information, select File > Export > Release Build. For more information, see [Using Adobe Flash Builder 4](#).

6 Verify that your new text appears in the TextArea control.

Create a linked resource to the LiveCycle Data Services configuration files

While working on the client-side of your applications, you often look at or change the LiveCycle Data Services configuration files. You can create a linked resource inside a Flash Builder project to make the LiveCycle Data Services configuration files easily accessible.

1 Right-click the project name in the project navigation view.

- 2 Select New > Folder in the pop-up menu.
- 3 Specify the name of the folder as it will appear in the navigation view. This name can be different from the name of the folder in the file system. For example, type **server-config**.
- 4 Click the Advanced button.
- 5 Select the Link To Folder In The File System option.
- 6 Click the Browse button and select the flex folder under the WEB-INF directory of your web application. For example, on a typical Windows installation that uses the Tomcat integrated server, select:
install_root/tomcat/webapps/lcds-samples/WEB-INF/flex.
- 7 Click Finish. The LiveCycle Data Services configuration files are now available in your Flash Builder project under the server-config folder.

Note: If you change anything in the services-config.xml file, you usually have to recompile your client-side applications and restart your server-side application for the changes to take effect.

Using mxmcl to compile client-side code

You use the mxmcl command line compiler to create SWF files from MXML, ActionScript, and other source files. Typically, you pass the name of the MXML application file to the compiler. The output is a SWF file. The mxmcl compiler ships in the bin directory of the Flex SDK. You run the mxmcl compiler as a shell script and executable file on Windows and UNIX systems. For more information, see the Flex documentation set.

The basic syntax of the mxmcl utility is as follows:

```
mxmcl [options] target_file
```

The target file of the compile is required. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file.

```
mxmcl -option arg1 arg2 arg3 -- target_file.mxml
```

To see a list of options for mxmcl, use the `help` option, as the following example shows:

```
mxmcl -help list
```

To see a list of all options available for mxmcl, including advanced options, use the following command:

```
mxmcl -help list advanced
```

The default output of mxmcl is *filename.swf*, where *filename* is the name of the target file. The default output location is in the same directory as the target, unless you specify an output file and location with the `output` option.

The mxmcl command line compiler does not generate an HTML wrapper. Create your own wrapper to deploy a SWF file that the mxmcl compiler produced. The wrapper embeds the SWF object in the HTML tag. The wrapper includes the `<object>` and `<embed>` tags, and scripts that support Flash Player version detection and history management. For information about creating an HTML wrapper, see the Flex Help Resource Center.

Note: Flash Builder automatically generates an HTML wrapper when you compile your application.

Compiling LiveCycle Data Services applications

Along with the standard options that you use with the mxmcl compiler, use the following options to specify information about your LiveCycle Data Services application.

- `services filename`
Specifies the location of the services-config.xml file.
- `library-path context-path`

Sets the library path of the LiveCycle Data Services SWC files. Use the `+ =` syntax with the library-path option to add the WEB-INF/flex/libs directory to the library path.

- `context-root context-path`

Sets the value of the context root of the application. This value corresponds to the `{context.root}` token in the services-config.xml file, which is often used in channel definitions. The default value is null.

Edit, compile, and deploy the Chat.mxml file

- 1 Unzip flex-src.zip in the lcds/tomcat/webapps/lcds-samples directory, as described in “[Running the LiveCycle Data Services sample applications](#)” on page 12.
- 2 Open the file `install_root/tomcat/webapps/lcds-samples/testdrive-chat/src/Chat.mxml` in an editor. Modify this path as necessary based on where you unzipped flex-src.zip.
- 3 Change the definition of the TextArea control so that it displays an initial text string when the application starts:
`<mx:TextArea id="log" width="100%" height="100%" text="My edited file!"/>`
- 4 Change the directory to `install_root/tomcat/webapps/lcds-samples`.
- 5 Use the following command to compile Chat.mxml:

Note: This command assumes that you added the mxmclc directory to your system path. The default location is `install_root/resources/flex_sdk/bin`.

```
mxmclc -strict=true
        -show-actionscript-warnings=true
        -use-network=true
        -services=WEB-INF/flex/services-config.xml
        -library-path+=WEB-INF/flex/libs
        -context-root=lcds-samples
        -output=testdrive-chat/Chat.swf
        testdrive-chat/src/Chat.mxml
```

The compiler writes the Chat.swf file to the lcds-samples/testdrive-chat directory.

- 6 Start the samples database and LiveCycle Data Services as described in “[Running the LiveCycle Data Services sample applications](#)” on page 12.
- 7 Request the application by using the URL `http://localhost:8400/lcds-samples/testdrive-chat/index.html`.
- 8 Verify that your new text appears in the TextArea control.

Rather than keep your source code in your deployment directory, you can set up a separate directory, and then copy Chat.swf to lcds-samples/testdrive-chat to deploy it.

Loading LiveCycle Data Services client applications as sub-applications

You can load sub-applications that use RPC services, the Message Service, or the Data Management Service into a parent client application. For general information about sub-applications, see *Creating and loading sub-applications* in the Flex documentation.

To load LiveCycle Data Services sub-applications into a parent application when using Flash Player 10.1 or later, use the SWFLoader control with its `loadForCompatibility` property to `true`. There is a single class table per security domain; setting the `loadForCompatibility` property to `true` lets you avoid class registration conflicts between sub-applications by ensuring that each application has its own class table and its own security domain.

There is a bug in Flash Player versions prior to version 10.1 where applications in a single physical domain also run in the same security domain and therefore only have one class table even when SWFLoader loadForCompatibility is set to true. For this reason, Adobe recommends that you use Flash Player 10.1 or later if your application is loading more than one sub-application that uses LiveCycle Data Services functionality.

If you must support Flash Player versions prior to version 10.1, there are a couple of workarounds to this issue. One workaround is to use the SWFLoader control with the loadForCompatibility property set to true, but load each sub-application from a different domain. You can accomplish this in a production system by using different subdomains for each SWF file, such as app1.domain.com/mylcdsapp.swf and app2.domain.com/mylcdsapp.swf, where each subdomain resolves to the same server or host. Because each sub-application is loaded from a different physical domain, Flash Player gives it a separate security domain.

The other workaround is to use the ModuleLoader control to load the sub-applications or use the SWFLoader control with the loadForCompatibility property set to false. In either case, in the parent application you define all data services and RPC classes as well as any strongly typed objects that your application uses. The following example shows this workaround with the SWFLoader control. FDSClasses and RPCClasses are the class definitions required for data services and RPC classes.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
backgroundColor="#FFFFFF">
    <mx:SWFLoader loadForCompatibility="false" id="s1"
        source="SampleDataService.swf"/>
    <mx:SWFLoader loadForCompatibility="false" id="s2"
        source="SampleDataService2.swf"/>
    <mx:Script>
        <![CDATA[
            FDSClasses;
            RPCClasses;
        ]]>
    </mx:Script>
</mx:Application>
```

Building your server-side application

You write the server-side part of a LiveCycle Data Services application in Java, and then use the javac compiler to compile it.

Creating a simple Java class to return data to the client

A common reason to create a server-side Java class is to represent data returned to the client. For example, the client-side RemoteObject component lets you access the methods of server-side Java objects to return data to the client.

The Test Drive sample application contains the Accessing Data Using Remoting sample where the client-side code uses the RemoteObject component to access product data on the server. The Product.java class represents that data. After starting the LiveCycle Data Services server and the samples database, view this example by opening the following URL in a browser: <http://localhost:8400/lcds-samples/testdrive-remoteobject/index.html>.

The source code for Product.java is in the *install_root/lcds-samples/WEB-INF/src/flex/samples/product* directory. For example, if you installed LiveCycle Data Services with the integrated Tomcat server on Microsoft Windows, the directory is *install_root/tomcat/webapps/lcds-samples/WEB-INF/src/flex/samples/product*.

Modify, compile, and deploy Product.java on the lcds-sample server

- 1 Start the samples database.

- 2 Start LiveCycle Data Services.
- 3 View the running example by opening the following URL in a browser:
<http://localhost:8400/lcds-samples/testdrive-remoteobject/index.html>
- 4 Click the Get Data button to download data from the server. Notice that the description column contains product descriptions.
- 5 In an editor, open the file *install_root/tomcat/webapps/lcds-samples/WEB-INF/src/flex/samples/product/Product.java*. Modify this path as necessary for your installation.
- 6 Modify the following `getDescription()` method definition so that it always returns the String "My description" rather than the value from the database:

```
public String getDescription() {  
    return description;  
}
```

The modified method definition appears as the following:

```
public String getDescription() {  
    return "My description.>";  
}
```

- 7 Change the directory to lcds-samples.
- 8 Compile Product.java by using the following javac command line:

```
javac -d WEB-INF/classes/ WEB-INF/src/flex/samples/product/Product.java
```

This command creates the file Product.class, and deploys it to the WEB-INF/classes/flex/samples/product directory.
- 9 View the running example by opening the following URL in a browser:
<http://localhost:8400/lcds-samples/testdrive-remoteobject/index.html>
- 10 Click the Get Data button.

Notice that the description column now contains the String "My description" for each product.

Creating a Java class that extends a LiveCycle Data Services class

As part of developing your server-side code, you can create a custom assembler class, factory class, or other type of Java class that extends the LiveCycle Data Services Java class library. For example, a *data adapter* is responsible for updating the persistent data store on the server in a manner appropriate to the specific data store type.

You perform many of the same steps to compile a Java class that extends the LiveCycle Data Services Java class library as you do for compiling a simple class. The major difference is to ensure that you include the appropriate LiveCycle Data Services JAR files in the classpath of your compilation so that the compiler can locate the appropriate files.

The Test Drive sample application contains the Data Management Service sample, in which the server-side code uses a custom assembler represented by the ProductAssembler.java class. To view the sample, start the LiveCycle Data Services server and the samples database. Then open the following URL in a browser: <http://localhost:8400/lcds-samples/testdrive-dataservice/index.html>.

The source code for ProductAssembler.java is in the directory *install_root/lcds-samples/WEB-INF/src/flex/samples/product*. For example, if you installed LiveCycle Data Services with the integrated Tomcat server on Microsoft Windows, the directory is *install_root/tomcat/webapps/lcds-samples/WEB-INF/src/flex/samples/product*.

Compile and deploy ProductAssembler.java on the lcds-sample server

- 1 View the running example by opening the following URL in a browser:

<http://localhost:8400/lcds-samples/testdrive-dataservice/index.html>

- 2 Click the Get Data button to download data from the server.
- 3 In an editor, open the file *install_root/tomcat/webapps/lcds-samples/WEB-INF/src/flex/samples/product/ProductAssembler.java*. Modify this path as necessary for your installation.
For more information on assemblers, see “[Standard assemblers](#)” on page 268.
- 4 Change the directory to *install_root/tomcat/webapps/lcds-samples/WEB-INF/src*.
- 5 Compile ProductAssembler.java by using the following javac command line:

```
javac
  -sourcepath .
  -d ..\classes
  -classpath ..\lib\flex-messaging-common.jar;
    ..\lib\flex-messaging-core.jar;
    ..\lib\flex-messaging-data-req.jar;
    ..\lib\flex-messaging-data.jar;
    ..\lib\flex-messaging-opt.jar;
    ..\lib\flex-messaging-proxy.jar;
    ..\lib\flex-messaging-remoting.jar
flex\samples\product\ProductAssembler.java
```

Notice that the classpath contains many, but not all, of the JAR files in WEB-INF/lib. If you are compiling other types of classes, you include additional JAR files in the classpath. These JAR files are also shipped in the *install_root/resources/lib* directory.

This command creates the file ProductAssembler.class, and deploys it to the WEB-INF/classes/flex/samples/product directory.

- 6 View the running example by opening the following URL in a browser:

<http://localhost:8400/lcds-samples/testdrive-dataservice/index.html>

- 7 Click the Get Data button to make sure that your code is working correctly.

Creating a combined Java and Flex project in Eclipse

If you use Eclipse for Java development, you can develop Java code in the same Eclipse project in which you develop Flex client code. You toggle between the Java and Flash perspectives to work on Java or Flex code. This procedure describes how to create a combined project. Another alternative is to use separate Eclipse projects for Java and Flex development.

Note: To create a combined Java and Flex project, use a plug-in configuration of Flash Builder. You must have the Eclipse Java perspective available in your installation.

- 1 Create a Java project in Eclipse.
- 2 When you create a Java project, you can optionally set the output location to be a linked directory in your LiveCycle Data Services web application. For example, you can save compiled POJO classes to the WEB-INF/classes directory of your web application.

Complete these steps to set the output location to a linked directory:

- a When configuring a project in the New Java Project dialog, select the Allow Output Folders For Source Folders option.

- b Click Browse to the right of the Default Output Folder field.
- c Select the project directory in the Folder Selection dialog.
- d Click Create New Folder.
- e Select Advanced > Link To Folder In File System.
- f Browse to the directory where you want to save output and click OK.
- g Click OK on the New Folder dialog.
- h Click OK on the Folder Selection dialog.
- i Click Finish on the New Java Project dialog.

Similarly, if you plan to export your output to a JAR file, you can save the JAR file to the WEB-INF/lib directory of your LiveCycle Data Services web application.

- 3 In the Package Explorer, right-click your new Java project and select Add/Change Project Type > Add Flex Project.
- 4 Set up your Flex project as described in “[Using Flash Builder to compile client-side code](#)” on page 15.

Debugging your application

If you encounter errors in your applications, you can use the debugging tools to perform the following:

- Set and manage breakpoints in your code
- Control application execution by suspending, resuming, and terminating the application
- Step into and over the code statements
- Select critical variables to watch
- Evaluate watch expressions while the application is running

Debugging Flex applications can be as simple as enabling `trace()` statements or as complex as stepping into a source files and running the code, one line at a time. The Flash Builder debugger and the command line debugger, `fdb`, let you step through and debug ActionScript files used by your Flex applications. For information on how to use the Flash Builder debugger, see *Using Adobe Flash Builder 4*. For more information on the command line debugger, `fdb`, see the Flex documentation set.

Using Flash Debug Player

To use the `fdb` command line debugger or the Flash Builder debugger, install and configure Flash Debug Player. To determine whether you are running the Flash Debug Player or the standard version of Flash Player, open any Flex application in Flash Player and right-click. If you see the Show Redraw Regions option, you are running Flash Debug Player. For more information about installing Flash Debug Player, see the LiveCycle Data Services installation instructions.

Flash Debug Player comes in ActiveX, Plug-in, and stand-alone versions for Microsoft Internet Explorer, Netscape-based browsers, and desktop applications. You can find Flash Debug Player installers in the following locations:

- Flash Builder: `install_dir/Player/os_version`
- Flex SDK: `install_dir/runtimes/player/os_version/`

Like the standard version of Adobe Flash Player 9, Flash Debug Player runs SWF files in a browser or on the desktop in a stand-alone player. Unlike Flash Player, the Flash Debug Player enables you to do the following:

- Output statements and application errors to the local log file of Flash Debug Player by using the `trace()` method.
- Write data services log messages to the local log file of Flash Debug Player.

- View run-time errors (RTEs).
- Use the fdb command line debugger.
- Use the Flash Builder debugging tool.
- Use the Flash Builder profiling tool.

Note: ADL logs `trace()` output from AIR applications.

Using logging to debug your application

One tool that can help in debugging is the logging mechanism. You can perform server-side and client-side logging of requests and responses.

Client-side logging

For client-side logging, you directly write messages to the log file, or configure the application to write messages generated by -Flex to the log file. Flash Debug Player has two primary methods of writing messages to a log file:

- The global `trace()` method. The global `trace()` method prints a String to the log file. Messages can contain checkpoint information to signal that your application reached a specific line of code, or the value of a variable.
- Logging API. The logging API, implemented by the `TraceTarget` class, provides a layer of functionality on top of the `trace()` method. For example, you can use the logging API to log debug, error, and warning messages generated by Flex while applications execute.

Flash Debug Player sends logging information to the `flashlog.txt` file. The operating system determines the location of this file, as the following table shows:

Operating system	Location of log file
Windows 95/98/ME/2000/XP	C:/Documents and Settings/username/Application Data/Macromedia/Flash Player/Logs
Windows Vista	C:/Users/username/AppData/Roaming/Macromedia/Flash Player/Logs
Mac OS X	/Users/username/Library/Preferences/Macromedia/Flash Player/Logs/
Linux	/home/username/.macromedia/Flash_Player/Logs/

Use settings in the `mm.cfg` text file to configure Flash Debug Player for logging. If this file does not exist, you can create it when you first configure Flash Debug Player. The location of this file depends on your operating system. The following table shows where to create the `mm.cfg` file for several operating systems:

Operating system	Location of mm.cfg file
Mac OS X	/Library/Application Support/Macromedia
Windows 95/98/ME	%HOMEDRIVE%/%HOMEPATH%
Windows 2000/XP	C:/Documents and Settings/username
Windows Vista	C:/Users/username
Linux	/home/username

The `mm.cfg` file contains many settings that you can use to control logging. The following sample `mm.cfg` file enables error reporting and trace logging:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

After you enable reporting and logging, call the `trace()` method to write a String to the `flashlog.txt` file, as the following example shows:

```
trace("Got to checkpoint 1.");
```

Insert the following MXML line to enable the logging of all Flex-generated debug messages to `flashlog.txt`:

```
<mx:TraceTarget loglevel="2" />
```

For information about client-side logging, see the Flex documentation set.

Server-side logging

You configure server-side logging in the logging section of the services configuration file, `services-config.xml`. By default, output is sent to `System.out`.

You set the logging level to one of the following available levels:

- All
- Debug
- Info
- Warn
- Error
- None

You typically set the server-side logging level to `Debug` to log all debug messages, and also all info, warning, and error messages. The following example shows a logging configuration that uses the `Debug` logging level:

```
<logging>

<!-- You may also use flex.messaging.log.ServletLogTarget. -->
<target class="flex.messaging.log.ConsoleTarget" level="Debug">
    <properties>
        <prefix>[Flex]</prefix>
        <includeDate>false</includeDate>
        <includeTime>false</includeTime>
        <includeLevel>false</includeLevel>
        <includeCategory>false</includeCategory>
    </properties>
    <filters>
        <pattern>Endpoint</pattern>
        <!--<pattern>Service.*</pattern>-->
        <!--<pattern>Message.*</pattern>-->
    </filters>
</target>
</logging>
```

For more information, see “[Logging](#)” on page 420.

Measuring application performance

As part of preparing your application for final deployment, you can test its performance to look for ways to optimize it. One place to examine performance is in the message processing part of the application. To help you gather this performance information, enable the gathering of message timing and sizing data.

The mechanism for measuring the performance of message processing is disabled by default. When enabled, information regarding message size, server processing time, and network travel time is captured. This information is available to the client that pushed a message to the server, to a client that received a pushed message from the server, or to a client that received an acknowledge message from the server in response to a pushed message. A subset of this information is also available for access on the server.

You can use this mechanism across all channel types, including polling and streaming channels, that communicate with the server. However, this mechanism does not work when you make a direct connection to an external server by setting the `useProxy` property to `false` for the `HTTPService` and `WebService` tags because it bypasses the LiveCycle Data Services Proxy Server.

You use two parameters in a channel definition to enable message processing metrics:

- `<record-message-times>`
- `<record-message-sizes>`

Set these parameters to `true` or `false`; the default value is `false`. You can set the parameters to different values to capture only one type of metric. For example, the following channel definition specifies to capture message timing information, but not message sizing information:

```
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf"
    class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
    <properties>
        <record-message-times>true</record-message-times>
        <record-message-sizes>false</record-message-sizes>
    </properties>
</channel-definition>
```

For more information, see “[Measuring message processing performance](#)” on page 121.

Deploying your application

Your production environment determines how you deploy your application. One option is to package your application and assets in a LiveCycle Data Services web application, and then create a single WAR file that contains the entire web application. You can then deploy the single WAR file on your production server.

Alternatively, you can deploy multiple LiveCycle Data Services applications on a single web application. In this case, your production server has an expanded LiveCycle Data Services web application to which you add the directories that are required to run your new application.

When you deploy your LiveCycle Data Services application in a production environment, ensure that you deploy all the necessary parts of the application, including the following:

- The compiled SWF file that contains your client-side application
- The HTML wrapper generated by Flash Builder or created manually if you use the mxmcl compiler
- The compiled Java class and JAR files that represent your server-side application
- Any run-time assets required by your application
- A LiveCycle Data Services web application
- Updated LiveCycle Data Services configuration files that contain the necessary information to support your application

Chapter 2: System architecture

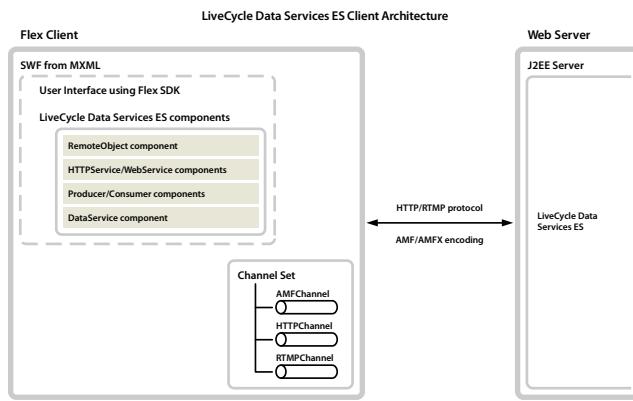
Client and server architecture

A LiveCycle Data Services application consists of a client application running in a web browser or Adobe AIR and a J2EE web application on the server that the client application communicates with. The client application can be a Flex application or it can be a combination of Flex, HTML, and JavaScript. When you use JavaScript, communication with the LiveCycle Data Services server is accomplished using the JavaScript proxy objects provided in the Ajax Client Library. For information about the Ajax Client Library, see “[The Ajax client library](#)” on page 460.

LiveCycle Data Services client architecture

LiveCycle Data Services clients use a message-based framework provided by LiveCycle Data Services to interact with the server. On the client side of the message-based framework are channels that encapsulate the connection behavior between the Flex client and the LiveCycle Data Services server. Channels are grouped together into channel sets that are responsible for channel hunting and channel failover. For information about client class APIs, see the *Adobe LiveCycle ActionScript Reference*.

The following illustration shows the LiveCycle Data Services client architecture:



Flex components

The following Flex components interact with a LiveCycle Data Services server:

- RemoteObject
- HTTPService
- WebService
- Producer
- Consumer
- DataService

All of these components, except for the DataService component, are included in the Flex SDK in the `rpc.swc` component library. The DataService component is included in the `fds.swc` component library provided in the LiveCycle Data Services installation.

Although the RemoteObject, Producer, and Consumer components are included with the Flex SDK, they require a server that can interpret the messages that they send. The BlazeDS and LiveCycle Data Services servers are two examples of such servers. A Flex application can also make direct HTTP service or web service calls to remote servers without LiveCycle Data Services in the middle tier. However, going through the LiveCycle Data Services Proxy Service is beneficial for several reasons; for more information, see “[Using RPC services](#)” on page 135.

Client-side components communicate with services on the LiveCycle Data Services server by sending and receiving messages of the correct type. For more information about messages, see “[Messages](#)” on page 28.

Channels and channel sets

A Flex component uses a channel to communicate with a LiveCycle Data Services server. A channel set contains channels; its primary function is to provide connectivity between the Flex client and the LiveCycle Data Services server. A channel set contains channels ordered by preference. The Flex component tries to connect to the first channel in the channel set and in the case where a connection cannot be established falls back to the next channel in the list. The Flex component continues to go through the list of channels in the order in which they are specified until a connection can be established over one of the channels or the list of channels is exhausted.

Channels encapsulate the connection behavior between the Flex components and the LiveCycle Data Services server. Conceptually, channels are a level below the Flex components and they handle the communication between the Flex client and the LiveCycle Data Services server. They communicate with their corresponding endpoints on the LiveCycle Data Services server; for more information about endpoints, see “[Endpoints](#)” on page 29.

Flex clients can use several different channel types, such as the AMFChannel, HTTPChannel, and RTMPChannel. Channel selection depends on a number of factors, including the type of application you are building. For example, if HTTP is the only protocol allowed in your environment, you would use the AMFChannel or HTTPChannel; you would not use the RTMPChannel, which uses the RTMP protocol. If non-binary data transfer is required, you would use the HTTPChannel, which uses a non-binary format called AMFX (AMF in XML). For more information about channels, see “[Channels and endpoints](#)” on page 38.

Messages

All communication between Flex client components and LiveCycle Data Services is performed with messages. Flex components use several message types to communicate with their corresponding services in LiveCycle Data Services. All messages have client-side (ActionScript) implementations and server-side (Java) implementations because the messages are serialized and deserialized on both the client and the server. You can also create messages directly in Java and have those messages delivered to clients using the server push API.

Some message types, such as AcknowledgeMessage and CommandMessage, are used across different Flex components and LiveCycle Data Services services. For example, to have a Producer component send a message to subscribed Consumer components, you create a message of type AsyncMessage and pass it to the `send()` method of the Producer component.

In other situations, you do not write code for constructing and sending messages. For example, you simply use a RemoteObject component to call the remote method from the Flex application. The RemoteObject component creates a RemotingMessage to encapsulate the RemoteObject call. In response it receives an AcknowledgeMessage from the server. The AcknowledgeMessage is encapsulated in a ResultEvent in the Flex application.

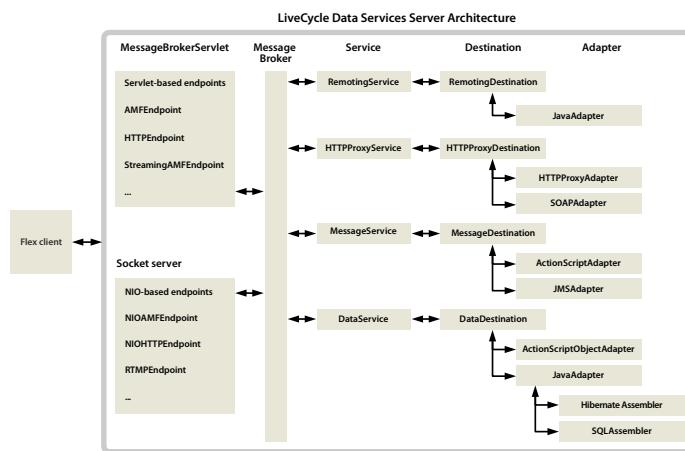
Sometimes you must create a message to send to the server. For example, you could send a message by creating an AsyncMessage and passing it to a Producer.

LiveCycle Data Services uses two patterns for sending and receiving messages: the request/reply pattern and the publish/subscribe pattern. RemoteObject, HTTPService, and WebService components use the request/reply message pattern, in which the Flex component makes a request and receives a reply to that request. Producer and Consumer components use the publish/subscribe message pattern. In this pattern, the Producer publishes a message to a destination defined on the LiveCycle Data Services server. All Consumers subscribed to that destination receive the message.

LiveCycle Data Services server architecture

The LiveCycle Data Services server is a combination of a J2EE web application and a highly scalable network socket server. A Flex client makes a request over a channel and the request is routed to an endpoint on the LiveCycle Data Services server. From the endpoint, the request is routed through a chain of Java objects that includes the MessageBroker object, a service object, a destination object, and finally an adapter object. The adapter fulfills the request either locally, or by contacting a backend system or a remote server such as Java Message Service (JMS) server.

The following illustration shows the LiveCycle Data Services server architecture:



Endpoints

LiveCycle Data Services has two types of endpoints: servlet-based endpoints and NIO-based endpoints.

Note: NIO-based endpoints are not available in BlazeDS.

NIO stands for Java New Input/Output. Servlet-based endpoints are inside the J2EE servlet container, which means that the servlet handles the I/O and HTTP sessions for the endpoints. Servlet-based endpoints are bootstrapped by the MessageBrokerServlet, which is configured in the web.xml file of the web application. In addition to the MessageBrokerServlet, an HTTP session listener is registered with the J2EE server in the web application's web.xml file so that LiveCycle Data Services has HTTP session attribute and binding listener support.

NIO-based endpoints run in an NIO-based socket server. These endpoints can offer significant scalability gains because they are not limited to one thread per connection and a single thread can manage multiple I/Os.

Flex client applications use channels to communicate with LiveCycle Data Services endpoints. There is a mapping between the channels on the client and the endpoints on the server. It is important that the channel and the endpoint use the same message format. A channel that uses the AMF message format, such as the AMFChannel, must be paired with an endpoint that also uses the AMF message format, such as the AMFEndpoint or the NIOAMFEndpoint. A channel that uses the AMFX message format such as the HTTPChannel cannot be paired with an endpoint that uses the AMF message format. Also, a channel that uses streaming must be paired with an endpoint that uses streaming.

You configure endpoints in the services-config.xml file in the WEB-INF/flex directory of your LiveCycle Data Services web application. For more information on servlet-based and NIO-based endpoints, see “[Channels and endpoints](#)” on page 38.

MessageBroker

The MessageBroker is responsible for routing messages to services and is at the core of LiveCycle Data Services on the server. After an endpoint initially processes the request, it extracts the message from the request and passes it to the MessageBroker. The MessageBroker inspects the message's destination and passes the message to its intended service. If the destination is protected by a security constraint, the MessageBroker runs the authentication and authorization checks before passing the message along (see “[Configuring security](#)” on page 432). You configure the MessageBroker in the services-config.xml file in the WEB-INF/flex directory of your LiveCycle Data Services web application.

Services and destinations

Services and destinations are the next links in the message processing chain in the LiveCycle Data Services server. The system includes four services and their corresponding destinations:

- RemotingService and RemotingDestination
- HTTPProxyService and HTTPProxyDestination
- MessageService and MessageDestination
- DataService and DataDestination

Note: *DataService and DataDestination are not available in BlazeDS.*

Services are the targets of messages from client-side Flex components. Think of destinations as instances of a service configured in a certain way. For example, a RemoteObject component is used on the Flex client to communicate with the RemotingService. In the RemoteObject component, you must specify a destination `id` property that refers to a remoting destination with certain properties, such as the class you want to invoke methods on. The mapping between client-side Flex components and LiveCycle Data Services services is as follows:

- HTTPSService and WebService communicate with HTTPProxyService/HTTPProxyDestination
- RemoteObject communicates with RemotingService/RemotingDestination
- Producer and Consumer communicate with MessageService/MessageDestination
- DataService communicates with DataService/DataDestination

You can configure services and their destinations in the services-config.xml file, but it is best practice to put them in separate files as follows:

- RemotingService configured in the remoting-config.xml file
- HTTPProxyService configured in the proxy-config.xml file
- MessageService configured in the messaging-config.xml file
- DataService configured in the data-management-config.xml file

More Help topics

[“Using RPC services” on page 135](#)

[“Using the Message Service” on page 190](#)

[“Introducing the Data Management Service” on page 221](#)

Adapters and assemblers

Adapters, and optionally assemblers, are the last link in the message processing chain. When a message arrives at the correct destination, it is passed to an adapter that fulfills the request either locally or by contacting a backend system or a remote server such as a JMS server. Some adapters use yet another layer, called assemblers, to further break down the processing. For example, a DataDestination can use JavaAdapter, and JavaAdapter could use HibernateAssembler to communicate with Hibernate. LiveCycle Data Services uses the following mappings between destinations and adapters/assemblers:

- RemotingDestination uses JavaAdapter
- HTTPProxyDestination uses HTTPProxyAdapter or SOAPAdapter
- MessageDestination uses ActionScriptAdapter or JMSAdapter
- DataDestination uses ASObjectAdapter or JavaAdapter
- JavaAdapter uses HibernateAssembler, SQLAssembler, or a custom assembler

Adapters and assemblers are configured along with their corresponding destinations in the same configuration files.

Although the LiveCycle Data Services server comes with a rich set of adapters and assemblers to communicate with different systems, custom adapters and assemblers can be plugged into the server. Similarly, you do not have to create all destinations in configuration files, but instead you can create them dynamically at server startup or when the server is running; for more information, see “[Run-time configuration](#)” on page 411.

For information about the LiveCycle Data Services server-side classes, see the Javadoc API documentation.

About configuration files

You configure LiveCycle Data Services in the services-config.xml file. The default location of this file is the WEB-INF/flex directory of your LiveCycle Data Services web application. You can set this location in the configuration for the MessageBrokerServlet in the WEB-INF/web.xml file.

You can include files that contain service definitions by reference in the services-config.xml file. Your LiveCycle Data Services installation includes the Remoting Service, Proxy Service, Message Service, and Data Management Service by reference.

The following table describes the typical setup of the configuration files. Commented versions of these files are available in the resources/config directory of the LiveCycle Data Services installation.

Filename	Description
services-config.xml	The top-level LiveCycle Data Services configuration file. This file usually contains security constraint definitions, channel definitions, and logging settings that each of the services can use. It can contain service definitions inline or include them by reference. Generally, the services are defined in the remoting-config.xml, proxy-config.xml, messaging-config.xml, and data-management-config.xml files.
remoting-config.xml	The Remoting Service configuration file, which defines Remoting Service destinations for working with remote objects. For information about configuring the Remoting Service, see “ Using RPC services ” on page 135.

Filename	Description
proxy-config.xml	The Proxy Service configuration file, which defines Proxy Service destinations for working with web services and HTTP services (REST services). For information about configuring the Proxy Service, see “ Using RPC services ” on page 135.
messaging-config.xml	The Message Service configuration file, which defines Message Service destinations for performing publish subscribe messaging. For information about configuring the Message Service, see “ Using the Message Service ” on page 190.
data-management-config.xml	The Data Management Service configuration file, which defines Data Management Service destinations. For information about configuring the Data Management Service, see “ Data Management Service configuration ” on page 244.

When you include a file by reference, the content of the referenced file must conform to the appropriate XML structure for the service. The file-path value is relative to the location of the services-config.xml file. The following example shows service definitions included by reference:

```
<services>
    <!-- REMOTING SERVICE -->
    <service-include file-path="remoting-config.xml"/>

    <!-- PROXY SERVICE -->
    <service-include file-path="proxy-config.xml"/>

    <!-- MESSAGE SERVICE -->
    <service-include file-path="messaging-config.xml"/>

    <!-- DATA MANAGEMENT SERVICE -->
    <service-include file-path="data-management-config.xml"/>
</services>
```

Configuration tokens

The configuration files sometimes contain special {server.name} and {server.port} tokens. These tokens are replaced with server name and port values based on the URL from which the SWF file is served when it is accessed through a web browser from a web server. Similarly, a special {context.root} token is replaced with the actual context root of a web application.

Note: If you use server tokens in a configuration file for an Adobe AIR application and you compile using that file, the application will not be able to connect to the server. You can avoid this issue by configuring channels in ActionScript rather than in a configuration file (see “[Channels and endpoints](#)” on page 38).

You can also use custom run-time tokens in service configuration files; for example, {messaging-channel} and {my.token}. You specify values for these tokens in Java Virtual Machine (JVM) options. The server reads these JVM options to determine what values are defined for them, and replaces the tokens with the specified values. If you have a custom token for which a value cannot be found, an error is thrown. Because {server.name}, {server.port}, and {context.root} are special tokens, no errors occur when these tokens are not specified in JVM options.

How you define JVM options depends on the application server you use. For example, in Apache Tomcat, you can define an environment variable JAVA_OPTS that contains tokens and their values, as this code snippet shows:

```
JAVA_OPTS=-Dmessaging.channel=my-amf -Dmy.token=myValue
```

Configuration elements

The following table describes the XML elements of the services-config.xml file. The root element is the `services-config` element.

XML element		Description
<code>services</code>		<p>Contains definitions of individual data services or references to other XML files that contain service definitions. It is a best practice to use a separate configuration file for each type of standard service. These services include the Proxy Service, Remoting Service, Message Service, and Data Management Service.</p> <p>The <code>services</code> element is declared at the top level of the configuration as a child of the root element, <code>services-config</code>. For information about configuring specific types of services, see the following topics:</p> <ul style="list-style-type: none"> • “Using RPC services” on page 135 • “Using the Message Service” on page 190 • “Data Management Service configuration” on page 244
<code>default-channels</code>		<p>Sets the application-level default channels to use for all services. The default channels are used when a channel is not explicitly referenced in a destination definition. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried.</p> <p>Default channels can also be defined individually for each service, in which case the application-level default channels are overwritten by the service level default channels. Application-level default channels are necessary when a dynamic component is created using the run-time configuration feature and no channel set has been defined for the component. In that case, application-level default channels are used to contact the destination.</p> <p>For more information about channels and endpoints, see “Channels and endpoints” on page 38.</p>
<code>service-include</code>		<p>Specifies the full path to an XML file that contains the configuration elements for a service definition.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>file-path</code> Path to the XML file that contains a service definition.
<code>service</code>		<p>Contains a service definition.</p> <p>(Optional) You can use the <code>service-include</code> element to include a file that contains a service definition by reference instead of inline in the <code>services-config.xml</code> file.</p> <p>In addition to standard data services, you can define custom bootstrap services here for use with the run-time configuration feature; bootstrap services dynamically create services, destinations, and adapters at server startup. For more information, see “Run-time configuration” on page 411.</p>
	<code>properties</code>	Contains service properties.
	<code>adapters</code>	Contains service adapter definitions that are referenced in a destination to provide specific types of functionality.

XML element			Description
			<p><code>adapter-definition</code></p> <p>Contains a service adapter definition. Each type of service has its own set of adapters that are relevant to that type of service. For example, Data Management Service destinations can use the Java adapter or the ActionScript object adapter. An <code>adapter-definition</code> has the following attributes:</p> <ul style="list-style-type: none"> • <code>id</code> Identifier of an adapter, which you use to reference the adapter inside a destination definition. • <code>class</code> Fully qualified name of the Java class that provides the adapter functionality. • <code>default</code> Boolean value that indicates whether this adapter is the default adapter for service destinations. The default adapter is used when you do not explicitly reference an adapter in a destination definition.
		<code>default-channels</code>	<p>Contains references to default channels. The default channels are used when a channel is not explicitly referenced in a destination definition. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried.</p>
			<p><code>channel</code></p> <p>Contains a reference to the <code>id</code> of a channel definition. A <code>channel</code> element contains the following attribute:</p> <ul style="list-style-type: none"> • <code>ref</code> The <code>id</code> value of a channel definition <p>For more information about channels and endpoints, see “Channels and endpoints” on page 38.</p>
		<code>destination</code>	<p>Contains a destination definition.</p>
			<p><code>adapter</code></p> <p>Contains a reference to a service adapter. If this element is omitted, the destination uses the default adapter.</p>
			<p><code>properties</code></p> <p>Contains destination properties.</p> <p>The properties available depend on the type of service, which the specified service class determines.</p>
			<p><code>channels</code></p> <p>Contains references to the channels that the service can use for data transport. The channels are tried in the order in which they are included in the file. When one is unavailable, the next is tried.</p> <p>The <code>channel</code> child element contains references to the <code>id</code> value of a channel. Channels are defined in the <code>channels</code> element at the top level of the configuration as a child of the root element, <code>services-config</code>.</p>

XML element		Description
		<p>security</p> <p>Contains a reference to a security constraint definition and login command definitions that are used for authentication and authorization.</p> <p>This element can also contain complete security constraint definitions instead of references to security constraints that are defined globally in the top-level <code>security</code> element.</p> <p>For more information, see “Security” on page 429.</p> <p>The <code>security-constraint</code> child element contains references to the <code>id</code> value of a security constraint definition or contains a security constraint definition.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>ref</code> The <code>id</code> value of a <code>security-constraint</code> element defined in the <code>security</code> element at the top level of the services configuration. • <code>id</code> Identifier of a security constraint when you define the actual security constraint in this element. <p>The <code>login-command</code> child element contains a reference to the <code>id</code> value of a login command definition that is used for performing authentication.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>ref</code> The <code>id</code> value of a login command definition.
security		<p>Contains security constraint definitions and login command definitions for authentication and authorization.</p> <p>For more information, see “Security” on page 429.</p>
	security-constraint	<p>Defines a security constraint.</p>
	login-command	<p>Defines a login command that is used for custom authentication.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>class</code> Fully qualified class name of a login command class. • <code>server</code> Application server on which custom authentication is performed. • <code>per-client-authentication</code> You can only set this attribute to <code>true</code> for a custom login command and not an application-server-based login command. Setting it to <code>true</code> allows multiple clients sharing the same session to have distinct authentication states. For example, two windows of the same web browser could authenticate users independently. This attribute is set to <code>false</code> by default.
channels		<p>Contains the definitions of message channels that are used to transport data between the server and clients.</p> <p>For more information about channels and endpoints, see “Channels and endpoints” on page 38.</p>

XML element			Description
	channel-definition		<p>Defines a message channel that can be used to transport data.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>id</code> Identifier of the channel definition. • <code>class</code> Fully qualified class name of a channel class.
		endpoint	<p>Specifies the endpoint URI and the endpoint class of the channel definition.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>uri</code> Endpoint URI • <code>class</code> Fully qualified name of the channel class used on the client.
		properties	<p>Contains the properties of a channel definition. The properties available depend on the type of channel specified.</p>
clusters			<p>Contains cluster definitions, which configure software clustering across multiple hosts.</p> <p>For more information, see “Clustering” on page 442.</p>
flex-client	reliable-reconnect-duration-millis		<p>Idle timeout for FlexClient state at the server, including reliable messaging sequences. To support reliable reconnects consistently across all supported channels and endpoints, this value must be defined and greater than 0.</p> <p>Any active sessions/connections keep idle FlexClient instances alive. This timeout only applies to instances that have no currently active associated sessions/connections.</p>
	heartbeat-interval-millis		<p>Optional setting that controls whether the client application issues periodic heartbeat requests to the server to keep an idle connection alive and to detect loss of connectivity. Use this setting to keep sessions alive.</p> <p>The default value of 0 disables this functionality. If the application sets this value, it should use a longer rather than shorter interval to avoid placing unnecessary load on the remote host.</p> <p>As an illustrative example, low-level TCP socket keep-alives generally default to an interval of 2 hours. That is a longer interval than most applications that enable heartbeats would likely want to use, but it serves as a clear precedent to prefer a longer interval over a shorter interval.</p>

XML element			Description
	timeout-minutes		<p>Each Flex application that connects to the server triggers the creation of a FlexClient instance that represents the remote client application. If the value of the <code>timeout-minutes</code> element is left undefined or set to 0 (zero), FlexClient instances on the server are shut down when all associated FlexSessions (corresponding to connections between the client and server) are shut down. If this value is defined, FlexClient instances are kept alive for this amount of idle time.</p> <p>FlexClient instances that have an associated RTMP connection/session open, are kept alive even if they are idle because the open connection indicates the remote client application is still running.</p> <p>For HTTP connections/sessions, if the remote client application is polling, the FlexClient is kept alive.</p> <p>If the remote client is not polling and a FlexClient instance is idle for this amount of time, it is shut down even if an associated HttpSession is still valid. This is because multiple Flex client applications can share a single HttpSession. A valid HttpSession does not indicate that a specific client application instance is still running.</p>
logging			Contains server-side logging configuration. For more information, see “ Logging ” on page 420.
	target		<p>Specifies the logging target class and the logging level.</p> <p>Attributes:</p> <ul style="list-style-type: none"> • <code>class</code> Fully qualified logging target class name. • <code>level</code> The logging level.
system			System-wide settings that do not fall into a previous category. In addition to locale information, it also contains redeployment and watch file settings.
	enforce-endpoint-validation		(Optional) Default value is <code>false</code> . When a destination is accessed over a channel, the message broker validates that the destination accepts requests over that channel. However, the client can disable this validation if the client has a null <code>id</code> value for its channel. When this setting is <code>true</code> , the message broker performs the endpoint validation even if the client channel has a null <code>id</code> value.
	locale		(Optional) Locale string; for example, “ <code>en</code> ”, “ <code>de</code> ”, “ <code>fr</code> ”, and “ <code>es</code> ” are valid locale strings.
	default-locale		<p>The default locale string.</p> <p>If no <code>default-locale</code> element is provided, a base set of English error messages is used.</p>
	redeploy		<p>Support for web application redeployment when configuration files are updated. This feature works with J2EE application server web application redeployment.</p> <p>The <code>touch-file</code> value is the file used by your application server to force web redeployment.</p> <p>Check the application server to confirm what the <code>touch-file</code> value should be.</p>
		enabled	Boolean value that indicates whether redeployment is enabled.

XML element			Description
		watch-interval	Number of seconds to wait before checking for changes to configuration files.
		watch-file	A data services configuration file watched for changes. The <code>watch-file</code> value must start with <code>{context.root}</code> or be an absolute path. The following example uses <code>{context.root}</code> : <code>{context.root}/WEB-INF/flex/data-management-config.xml</code>
		touch-file	The file that the application server uses to force redeployment of a web application. The value of the <code>touch-file</code> element must start with <code>{context.root}</code> or be an absolute path. Check the application server documentation to determine the <code>touch-file</code> value. For Tomcat, the <code>touch-file</code> value is the <code>web.xml</code> file, as the following example shows: <code>{context.root}/WEB-INF/web.xml</code>

Channels and endpoints

Channels are the client-side representation of the connection to a service, while endpoints are the server-side representation.

About channels and endpoints

Channels are client-side objects that encapsulate the connection behavior between Flex components and the LiveCycle Data Services server or the Edge Server. Channels communicate with corresponding endpoints on the LiveCycle Data Services server. You configure the properties of a channel and its corresponding endpoint in the `services-config.xml` file.

For more information on the Edge Server, see “[Edge Server](#)” on page 391.

How channels are assigned to a Flex component

Flex components use channel sets, which contain one or more channels, to contact the server. You can automatically or manually create and assign a channel set to a Flex component. The channel allows the component to contact the endpoint, which forwards the request to the destination.

If you compile an MXML file using the MXML compiler option `-services` pointing to the `services-config.xml` file, the component (RemoteObject, HTTPService, and so on) is automatically assigned a channel set that contains one or more appropriately configured channel instances. The configuration is based on the channel definition assigned to a destination in a configuration file. Alternatively, if you do not compile your application with the `-services` option or want to override the compiled-in behavior, you can create a channel set in MXML or ActionScript, populate it with one or more channels, and then assign the channel set to the Flex component.

Application-level default channels are especially important when you want to use dynamically created destinations and you do not want to create and assign channel sets to your Flex components that use the dynamic destination. In that case, application-level default channels are used. For more information, see “[Assigning channels and endpoints to a destination](#)” on page 42.

When you compile a Flex client application with the MXML compiler-`-services` option, it contains all of the information from the configuration files that is needed for the client to connect to the server.

Configuring channels and endpoints

You can configure channels in channel definitions in the services-config.xml file or on the Flex client.

Configuring channels on the server

The channel definition in the following services-config.xml file snippet creates an AMFChannel that communicates with an AMFEndpoint on the server:

```
<channels>
  ...
<channel-definition id="samples-amf"
    type="mx.messaging.channels.AMFChannel">
  <endpoint url="http://servername:8400/myapp/messagebroker/amf" port="8700"
    type="flex.messaging.endpoints.AMFEndpoint"/>
</channel-definition>
</channels>
```

The `channel-definition` element specifies the following information:

- `id` and channel class `type` of the client-side channel that the Flex client uses to contact the server
- `remote` attribute that specifies that the endpoint is remote. In that case, the endpoint is not started on this server and it is assumed that the client will connect to a remote endpoint on another server. This is useful when the client is compiled against this configuration but some of the endpoints are on a remote server.
- `endpoint` element that contains the URL and endpoint class type of the server-side endpoint
- `properties` element that contains channel and endpoint properties
- `server` element, which when using an NIO-based channel and endpoint optionally refers to a shared NIO server configuration

The endpoint URL is the specific network location that the endpoint is exposed at. The channel uses this value to connect to the endpoint and interact with it. The URL must be unique across all endpoints exposed by the server. The `url` attribute can point to the MessageBrokerServlet or an NIO server if you are using an NIO-based endpoint.

Configuring channels on the client

To create channels at runtime in Flex code, you create your own channel set in MXML or ActionScript, add channels to it, and then assign the channel set to a component. This process is common in the following situations:

- You do not compile your MXML file using the `-services` MXML compiler option. This is useful when you do not want to hard code endpoint URLs into your compiled SWF files on the client. It is also useful when you want to use a remote server when developing an application in Flash Builder.
- You want to use a dynamically created destination (the destination is not in the services-config.xml file) with the run-time configuration feature. For more information, see “[Run-time configuration](#)” on page 411.
- You want to control in your client code the order of channels that a Flex component uses to connect to the server.

When you create and assign a channel set on the client, the client requires the correct channel type and endpoint URL to contact the server. The client does not specify the endpoint class that handles that request, but there must be a channel definition in the services-config.xml file that specifies the endpoint class to use with the specified endpoint URL.

The following example shows a RemoteObject component that defines a channel set and channel inline in MXML:

```
...
<RemoteObject id="ro" destination="Dest">
    <mx:channelSet>
        <mx:ChannelSet>
            <mx:channels>
                <mx:AMFChannel id="myAmf"
                    uri="http://myserver:2000/myapp/messagebroker/amf"/>
            </mx:channels>
        </mx:ChannelSet>
    </mx:channelSet>
</RemoteObject>
...
```

The following example shows ActionScript code that is equivalent to the MXML code in the previous example:

```
...
private function run():void {
    ro = new RemoteObject();
    var cs:ChannelSet = new ChannelSet();
    cs.addChannel(new AMFChannel("myAmf",
        "http://myserver:2000/eqa/messagebroker/amf"));
    ro.destination = "Dest";
    ro.channelSet = cs;
}
...
```

Important: When you create a channel on the client, you still must include a channel definition that specifies an endpoint class in the services-config.xml file. Otherwise, the message broker cannot pass a Flex client request to an endpoint.

To further externalize configuration, you can pass the endpoint URL value to the client at runtime. One way to do this is by reading a configuration file with an HTTPService component at application startup. The configuration file includes the information to programmatically create a channel set at runtime. You can use E4X syntax to get information from the configuration file.

The following MXML application shows this configuration file technique:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="configSrv.send() " >
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.messaging.channels.AMFChannel;
            import mx.messaging.ChannelSet;
            import mx.rpc.events.ResultEvent;
            private var channelSet:ChannelSet;
            private function configResultHandler(event:ResultEvent):void
            {
                var xml:XML = event.result as XML;
                var amfEndpoint:String = "" + xml..channel.(@id=="amf").@endpoint;
                if (amfEndpoint == "")
                {
                    Alert.show("amf channel not configured", "Error");
                }
                else
                {
                    channelSet = new ChannelSet();
                    var channel:AMFChannel = new AMFChannel("my-amf", amfEndpoint);
                    channelSet.addChannel(channel);
                    ro.channelSet = channelSet;
                    ro.getProducts();
                }
            }
        ]]>
    </mx:Script>
    <mx:HTTPService id="configSrv" url="config.xml" resultFormat="e4x"
        result="configResultHandler(event)"/>
    <mx:RemoteObject id="ro" destination="product"/>
    <mx:DataGrid dataProvider="{ro.getProducts.lastResult}" width="100%" height="100%"/>
</mx:Application>

```

The MXML application reads the following configuration file:

```

<?xml version="1.0" encoding="utf-8"?>
<config>
    <channels>
        <channel id="amf" endpoint="http://localhost:8400/lcds-samples/messagebroker/amf"/>
    </channels>
</config>

```

Configuring a Flash Builder project with client-configured channels

When you create a new LiveCycle Data Services or BlazeDS project in Flash Builder, you typically select J2EE as the Application Server Type and then check Use Remote Object Access Service. This adds an MXML compiler argument that points to the services-config.xml file. If you check the Flex Compiler properties of your Flash Builder project, you see something like this:

```
-services "c:\lcds\tomcat\webapps\samples\WEB-INF\flex\services-config.xml"
```

When you compile the application, the required values of the services-config.xml are included in the SWF file. The services-config.xml file is read at compile time and not at runtime as you might assume. You can use tokens such as {server.name}, {server.port}, and {context.root} in the services-config.xml file. The {context.root} token is substituted at compile time. The {server.name} and {server.port} are replaced at runtime with the server name and port number of the server the SWF is loaded from; you can't use these tokens for AIR applications.

When you configure channels on the client, you can avoid dependency on the services-config.xml file. You can create a new Flex project with no Application Server Type settings because the channels are configured at runtime in the SWF file. For existing LiveCycle Data Services or BlazeDS projects, you can remove the services-config.xml compiler argument.

Assigning channels and endpoints to a destination

Settings in the LiveCycle Data Services configuration files determine the channels and endpoints from which a destination can accept messages, invocations, or data, except when you use the run-time configuration feature. The channels and endpoints are determined in one of the following ways:

- If most of the destinations across all services use the same channels, you can define application-level default channels in the services-config.xml file, as the following example shows.

Note: Using application-level default channels is a best practice whenever possible.

```
<services-config ...>
...
<default-channels>
    <channel ref="my-http"/>
    <channel ref="my-amf"/>
</default-channels>
...
```

In this case, all destinations that do not define channels use a default channel. You can override the default channel setting by specifying a channel at the destination level or service level.

- If most of the destinations in a service use the same channels, you can define service-level default channels, as the following example shows:

```
<service ...>
...
<default-channels>
    <channel ref="my-http"/>
    <channel ref="my-amf"/>
</default-channels>
...
```

In this case, all destinations in the service that do not explicitly specify their own channels use the default channel.

- The destination definition can reference a channel inline, as the following example shows:

```
<destination id="sampleVerbose">
    <channels>
        <channel ref="my-secure-amf"/>
    </channels>
...
</destination>
```

Fallback and failover behavior

The primary reason that channels are contained in a channel set is to provide a fallback mechanism from one channel to the next listed in the channel set, and so on, in case the first choice is unable to establish a connection to the server. For example, you could define a channel set that falls back from a StreamingAMFChannel to an AMFChannel with polling enabled to work around network components such as web server connectors, HTTP proxies, or reverse proxies that could buffer chunked responses incorrectly. You can also use this functionality to provide different protocol options so that a client can first try to connect using RTMP and if that fails, can fall back to HTTP.

The connection process involves searching for the first channel and trying to connect to it. In addition to the fallback behavior that the channel set provides, the channel defines a `failoverURIs` property. This property lets you configure a channel in ActionScript that causes failover across this array of endpoint URLs when it tries to connect to its destination. If the channel defines failover URIs, each is attempted before the channel gives up and the channel set searches for the next available channel. If no channel in the set can connect, any pending unsent messages generate faults on the client.

If the channel was successfully connected before experiencing a fault or disconnection, it attempts a pinned reconnection to the same endpoint URL once. If this immediate reconnection attempt fails, the channel falls back to its previous failover strategy and attempts to fail over to other server nodes in the cluster or fall back to alternate channel protocols.

Choosing an endpoint

The two types of endpoints in LiveCycle Data Services are servlet-based endpoints and NIO-based endpoints that use Java New I/O APIs.

Note: *NIO-based endpoints are not available in BlazeDS.*

The J2EE servlet container manages networking, IO, and HTTP session maintenance for the servlet-based endpoints. NIO-based endpoints are outside the servlet container and run inside an NIO-based socket server. NIO-based endpoints can offer significant scalability gains. Because they are NIO-based, they are not limited to one thread per connection. Far fewer threads can efficiently handle high numbers of connections and IO operations. If the web application is not servicing general servlet requests, you can configure the servlet container to bind non-standard HTTP and HTTPS ports. Ports 80 and 443 are then free for your NIO-based endpoints to use.

You can also deploy an Edge Server in your DMZ to forward client requests to a LiveCycle Data Services server in the application tier. In this configuration, clients communicate with the Edge Server in the DMZ, and not directly with the LiveCycle Data Services server inside the internal firewall. This configuration gives you more options for the endpoints you can use and the ports those endpoints can listen over. For more information about the Edge Server, see “[Edge Server](#)” on page 391.

The servlet-based endpoints are part of both BlazeDS and LiveCycle Data Services. Reasons to use servlet-based endpoints when you have LiveCycle Data Services are that you must include third-party servlet filter processing of requests and responses or you must access data structures in the application server HttpSession.

The NIO-based endpoints include RTMP endpoints as well as NIO-based AMF and HTTP endpoints that use the same client-side channels as their servlet-based counterparts.

The following situations prevent the NIO socket server used with NIO-based endpoints from creating a real-time connection between client and server in a typical deployment of a rich Internet application:

- The only client access to the Internet is through a proxy server.
- The application server on which LiveCycle Data Services is installed can only be accessed from behind the web tier in the IT infrastructure.

Servlet-based streaming endpoints and long polling are good alternatives when NIO-based streaming is not an option. In the worst case, the client falls back to simple polling. The main disadvantages of polling are increased overhead on client and server machines, and increased network latency.

Servlet-based channel and endpoint combinations

LiveCycle Data Services provides the following servlet-based channel and endpoint combinations. A secure version of each of these channels/endpoints transports data over a secure HTTPS connection. The names of the secure channels and endpoints all start with the text "Secure"; for example, SecureAMFChannel and SecureAMFEndpoint.

Servlet-based channel/endpoint classes	Description
AMFChannel/AMFEndpoint	A simple channel/endpoint that transports data over HTTP in the binary AMF format in an asynchronous call and response model. Use for RPC requests/responses with RPC-style Flex components such as RemoteObject, HTTPService, and WebService. You can also configure a channel that uses this endpoint to repeatedly poll the endpoint for new messages. You can combine polling with a long wait interval for long polling, which handles near real-time communication. For more information, see " Simple channels and endpoints " on page 47.
HTTPChannel/HTTPEndpoint	Provides the same behavior as the AMF channel/endpoint, but transports data in the AMFX format, which is the text-based XML representation of AMF. Transport with this endpoint is not as fast as with the AMFEndpoint because of the overhead of text-based XML. Use when binary AMF is not an option in your environment. For more information, see " Simple channels and endpoints " on page 47.
StreamingAMFChannel/StreamingAMFEndpoint	Streams data in real time over the HTTP protocol in the binary AMF format. Use for real-time data services, such as the Data Management Service and the Message Service where streaming data is critical to performance. For more information, see " Streaming AMF and HTTP channels " on page 52.
StreamingHTTPChannel/StreamingHTTPEndpoint	Provides the same behavior model as the streaming AMF channel/endpoint, but transports data in the AMFX format, which is the text-based XML representation of AMF. Transport with this endpoint is not as fast as with the StreamingAMFEndpoint because of the overhead of text-based XML. Use when binary AMF is not an option in your environment. For more information, see " Streaming AMF and HTTP channels " on page 52.

NIO-based channel and endpoint combinations

The NIO-based endpoints include RTMP endpoints as well as NIO-based AMF and HTTP endpoints that use the same client-side channels as their servlet-based counterparts.

Note: NIO-based endpoints are not available in Blazeds.

LiveCycle Data Services provides the following NIO-based channels/endpoints. A secure version of the RTMP channel/endpoint transports data over an RTMPS connection. A secure version of each of the HTTP and AMF channels/endpoints transports data over an HTTPS connection. The names of the secure channels and endpoints all start with the text "Secure"; for example, SecureAMFChannel and SecureNIOAMFEndpoint.

NIO-based channel/endpoint classes	Description
RTMPChannel/RTMPEndpoint	<p>Streams data in real time over the TCP-based RTMP protocol in the binary AMF format. Use for real-time data services, such as the Data Management Service and the Message Service where streaming data is critical to performance.</p> <p>The RTMP channel/endpoint uses an NIO server to support scaling up to thousands of connections. It uses a single duplex socket connection to the server and gives the server the best notification of Flash Player being shut down. If the direct connect attempt fails, Flash Player attempts a CONNECT tunnel through an HTTP proxy if the browser defines one (resulting in a direct, tunneled duplex socket connection to the server). In the worst case, Flash Player falls back to adaptive polling of HTTP requests that tunnel RTMP data back and forth between client and server, or it fails to connect entirely.</p> <p>When you define an RTMP endpoint with a URI that starts with <code>rtmp:</code> and specifies no port, the endpoint automatically binds ports 1935 and 80 when the server starts.</p> <p>When you define an RTMP endpoint with a URI that starts with <code>rtmpt:</code> and specifies no port, the endpoint will automatically bind port 80 when the server starts.</p> <p>When you define a secure RTMP endpoint that specifies no port, the endpoint automatically binds port 443 when the server starts.</p> <p>A defined (hardcoded) port value in the channel/endpoint URI overrides these defaults, and a <code>bind-port</code> configuration setting overrides these as well. When using a defined port in the URI or <code>bind-port</code> the endpoint binds just that single port at startup.</p> <p>For more information, see “Configuring channels with NIO-based endpoints” on page 55.</p>
AMFChannel/NIOAMFEndpoint	NIO-based version of the AMF channel/endpoint. Uses an NIO server and a minimal HTTP stack to support scaling up to thousands of connections.
StreamingAMFChannel/NIOSreamingAMFEndpoint	NIO-based version of streaming AMF channel/ endpoint. Uses an NIO server and a minimal HTTP stack to support scaling up to thousands of connections.
HTTPChannel/NIOHTTPEndpoint	NIO-based version of HTTP channel/endpoint. Uses an NIO server and a minimal HTTP stack to support scaling up to thousands of connections.
StreamingHTTPChannel/StreamingNIOHTTPEndpoint	NIO-based version of streaming HTTP channel/endpoint. Uses an NIO server and a minimal HTTP stack to support scaling up to thousands of connections.

Choosing a channel

Depending on your application requirements, you can use simple AMF or HTTP channels without polling or with piggybacking, polling, or long polling. You can also use streaming RTMP, AMF, or HTTP channels. The difference between AMF and HTTP channels is that AMF channels transport data in the binary AMF format and HTTP channels transport data in AMFX, the text-based XML representation of AMF. Because AMF channels provide better performance, use an HTTP channel instead of an AMF channel only when you have auditing or compliance requirements that preclude the use of binary data over your network or when you want the contents of messages to be easily readable over the network (on the wire).

Non-polling AMF and HTTP channels

You can use AMF and HTTP channels without polling for remote procedure call (RPC) services, such as remoting service calls, proxied HTTP service calls and web service requests, or Data Management Service requests without automatic synchronization. These scenarios do not require the client to poll for messages or the server to push messages to the client.

Piggybacking on AMF and HTTP channels

The piggybacking feature enables the transport of queued messages along with responses to any messages the client sends to the server over the channel. By default, piggybacking is disabled. You can set it to `true` in the `piggybacking-enabled` property of a channel definition in the `services-config.xml` file or in the `piggybackingEnabled` property of a Channel instance when you create a channel in ActionScript.

Piggybacking provides lightweight pseudo polling, where rather than the client channel polling the server on a fixed or adaptive interval, when the client sends a non-command message to the server (using a Producer, RemoteObject, or DataService object), the server sends any pending data for client messaging or data management subscriptions along with the response to the client message.

Piggybacking can also be used on a channel that has polling enabled but on a wide interval like 5 seconds or 10 seconds or more, in which case the application appears more responsive if the client is sending messages to the server. In this mode, the client sends a poll request along with any messages it sends to the server between its regularly scheduled poll requests. The channel piggybacks a poll request along with the message being sent, and the server piggybacks any pending messages for the client along with the acknowledge response to the client message.

Polling AMF and HTTP channels

AMF and HTTP channels support simple polling mechanisms that clients can use to request messages from the server at set intervals. A polling AMF or HTTP channel is useful when other options such as long polling or streaming channels are not acceptable and also as a fallback channel when a first choice, such as a streaming channel, is unavailable at run time.

Long polling AMF and HTTP channels

You can use AMF and HTTP channels in long polling mode to get pushed messages to the client when the other more efficient and real-time mechanisms are not suitable. This mechanism uses the normal application server HTTP request processing logic and works with typical J2EE deployment architectures.

You can establish long polling for any channel that uses a non-streaming AMF or HTTP endpoint by setting the `polling-enabled`, `polling-interval-millis`, `wait-interval-millis`, and `client-wait-interval-millis` properties in a channel definition; for more information, see “[Simple channels and endpoints](#)” on page 47.

Streaming channels

For streaming, you can use RTMP channels, or streaming AMF or HTTP channels. Streaming channels must be paired with corresponding streaming endpoints that are NIO-based or servlet-based. Streaming AMF and HTTP channels work with servlet-based streaming AMF or HTTP endpoints or NIO-based streaming AMF or HTTP endpoints. RTMP channels work with NIO-based RTMP endpoints.

For more information about endpoints, see “[Choosing an endpoint](#)” on page 43.

Configuring channels with servlet-based endpoints

The servlet-based endpoints are part of both BlazeDS and LiveCycle Data Services.

Simple channels and endpoints

The AMFEndpoint and HTTPEndpoint are simple servlet-based endpoints. You generally use channels with these endpoints without client polling for RPC service components, which require simple call and response communication with a destination. When working with the Message Service or Data Management Service, you can use these channels with client polling to constantly poll the destination on the server for new messages, or with long polling to provide near real-time messaging when using a streaming channel is not an option in your network environment.

Property	Description
polling-enabled	Optional channel property. Default value is false.
polling-interval-millis	Optional channel property. Default value is 3000. This parameter specifies the number of milliseconds the client waits before polling the server again. When <code>polling-interval-millis</code> is 0, the client polls as soon as it receives a response from the server with no delay.
wait-interval-millis	<p>Optional endpoint property. Default value is 0. This parameter specifies the number of milliseconds the server poll response thread waits for new messages to arrive when the server has no messages for the client at the time of poll request handling. For this setting to take effect, you must use a nonzero value for the <code>max-waiting-poll-requests</code> property.</p> <p>A value of 0 means that server does not wait for new messages for the client and returns an empty acknowledgment as usual. A value of -1 means that server waits indefinitely until new messages arrive for the client before responding to the client poll request.</p> <p>The recommended value is 60000 milliseconds (one minute).</p>
client-wait-interval-millis	<p>Optional channel property. Default value is 0. Specifies the number of milliseconds the client will wait after it receives a poll response from the server that involved a server wait (<code>wait-interval-millis</code> is set to a non-zero value).</p> <p>A value of 0 means the client uses its configured <code>polling-interval-millis</code> value to determine the wait until its next poll. Otherwise, this value overrides the default polling interval of the client.</p> <p>Setting this value to 1 allows clients that poll the server with <code>wait</code> (<code>wait-interval-millis</code> is set to a non-zero value) to poll immediately upon receiving a poll response from the server, providing a real-time message stream from the server to the client.</p> <p>Clients that poll the server and are not serviced with a server wait (<code>wait-interval-millis</code> is not set to a non-zero value) use the <code>polling-interval-millis</code> value.</p>
max-waiting-poll-requests	Optional endpoint property. Default value is 0. Specifies the maximum number of server poll response threads that can be in wait state. When this limit is reached, the subsequent poll requests are treated as having zero <code>wait-interval-millis</code> .
piggybacking-enabled	Optional endpoint property. Default value is <code>false</code> . Enable to support piggybacking of queued messaging and data management subscription data along with responses to any messages the client sends to the server over this channel.

Property	Description
login-after-disconnect	<p>Optional channel property. Default value is <code>false</code>. Setting to <code>true</code> causes clients to automatically attempt to reauthenticate themselves with the server when they send a message that fails because credentials have been reset due to server session timeout. The failed messages are resent after reauthentication, making the session timeout transparent to the client with respect to authentication.</p> <p><code>login-after-disconnect</code> functionality only works for requests (messages) that fault due to an authentication error for a client that was previously authenticated but is no longer authenticated. The destination or service must be protected with a security constraint. The <code>login-after-disconnect</code> setting is not used if a message faults for any other reason.</p>
flex-client-outbound-queue-processor	<p>Optional channel property. Use to manage messaging quality of service for subscribers. Every client that subscribes to the server over this channel is assigned a unique instance of the specified outbound queue processor implementation that manages the flow of messages to the client. This can include message conflation, filtering, scheduled delivery and load shedding. You can define configuration properties, and if so, they are used to configure each new queue processor instance that is created. The following example shows how to provide a configuration property:</p> <pre data-bbox="706 983 1148 1108"><flex-client-outbound-queue-processor class="my.company.QoSQueueProcessor"> <properties> <custom-property>5000</custom-property> </properties> </flex-client-outbound-queue-processor></pre>
serialization	<p>Optional serialization properties on endpoint. For more information, see “Configuring AMF serialization on a channel” on page 84.</p>
connect-timeout-seconds	<p>Optional channel property. Default value is 0. Use to limit the client channel's connect attempt to the specified time interval.</p>
invalidate-session-on-disconnect	<p>Optional endpoint property. Disabled by default. If enabled, when a disconnect message is received from a client channel, the corresponding server session is invalidated. If the client is closed without first disconnecting its channel, no disconnect message is sent, and the server session is invalidated when its idle timeout elapses.</p>
add-no-cache-headers	<p>Optional endpoint property. Default value is <code>true</code>. HTTPS requests on some browsers do not work when pragma no-cache headers are set. By default, the server adds headers, including pragma no-cache headers to HTTP responses to stop caching by the browsers.</p>

Non-polling AMF and HTTP channels

The simplest types of channels are AMF and HTTP channels in non-polling mode, which operate in a single request-reply pattern. The following example shows AMF and HTTP channel definitions configured for no polling:

```
<!-- Simple AMF -->
<channel-definition id="samples-amf"
    type="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8100/myapp/messagebroker/amf"
        type="flex.messaging.endpoints.AmfEndpoint"/>
</channel-definition>
<!-- Simple secure AMF -->
<channel-definition id="my-secure-amf"
    class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://{server.name}:9100/dev/messagebroker/
        amfsecure" class="flex.messaging.endpoints.SecureAMFEndpoint"/>
</channel-definition>
<!-- Simple HTTP -->
<channel-definition id="my-http"
    class="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://{server.name}:8100/dev/messagebroker/http"
        class="flex.messaging.endpoints.HTTPEndpoint"/>
</channel-definition>
<!-- Simple secure HTTP -->
<channel-definition id="my-secure-http" class="mx.messaging.channels.SecureHTTPChannel">
    <endpoint url=
        "https://{server.name}:9100/dev/messagebroker/
        httpssecure"
        class="flex.messaging.endpoints.SecureHTTPEndpoint"/>
</channel-definition>
```

Polling AMF and HTTP channels

You can use an AMF or HTTP channel in polling mode to repeatedly poll the endpoint to create client-pull message consumers. The interval at which the polling occurs is configurable on the channel. You can also manually poll by calling the `poll()` method of a channel for which polling is enabled; for example, you want set the polling interval to a high number so that the channel does not automatically poll, and call the `poll()` method to poll manually based on an event, such as a button click.

When you use a polling AMF or HTTP channel, you set the `polling` property to `true` in the channel definition. You can also configure the polling interval in the channel definition.

Note: You can also use AMF and HTTP channels in long polling mode to get pushed messages to the client when the other more efficient and real-time mechanisms are not suitable. For information about long polling, see “[Long polling AMF and HTTP channels](#)” on page 50.

The following example shows AMF and HTTP channel definitions configured for polling:

```
<!-- AMF with polling -->
<channel-definition id="samples-polling-amf"
    class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8700/dev/messagebroker/amfpolling"
        type="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>8000</polling-interval-millis>
    </properties>
</channel-definition>
<!-- HTTP with polling -->
<channel-definition id="samples-polling-http"
    class="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://{server.name}:8700/dev/messagebroker/httppolling"
        type="flex.messaging.endpoints.HTTPEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>8000</polling-interval-millis>
    </properties>
</channel-definition>
```

Note: You can also use secure AMF or HTTP channels in polling mode.

Long polling AMF and HTTP channels

In the default configuration for a polling AMF or HTTP channel, the endpoint does not wait for messages on the server. When the poll request is received, it checks whether any messages are queued for the polling client and if so, those messages are delivered in the response to the HTTP request. You configure long polling in the same way as polling, but you also must set the `wait-interval-millis`, `max-waiting-poll-requests`, and `client-wait-interval-millis` properties.

To achieve long polling, you set the following properties in the `properties` section of a channel definition in the `services-config.xml` file:

- `polling-enabled`
- `polling-interval-millis`
- `wait-interval-millis`
- `max-waiting-poll-requests`.
- `client-wait-interval-millis`

The following example shows AMF and HTTP channel definitions configured for long polling:

```

<!-- Long polling AMF -->
<channel-definition id="my-amf-longpoll" class="mx.messaging.channels.AMFChannel">
    <endpoint
        url="http://servername:8700/contextroot/messagebroker/myamflongpoll"
        class="flex.messaging.endpoints.AMFEEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>0</polling-interval-millis>
        <wait-interval-millis>60000</wait-interval-millis>
        <client-wait-interval-millis>3000</client-wait-interval-millis>
        <max-waiting-poll-requests>100</max-waiting-poll-requests>
    </properties>
</channel-definition>

<!-- Long polling HTTP -->
<channel-definition id="my-http-longpoll" class="mx.messaging.channels.HTTPChannel">
    <endpoint
        url="http://servername:8700/contextroot/messagebroker/myhttplongpoll"
        class="flex.messaging.endpoints.HTTPPEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>0</polling-interval-millis>
        <wait-interval-millis>60000</wait-interval-millis>
        <client-wait-interval-millis>3000</client-wait-interval-millis>
        <max-waiting-poll-requests>100</max-waiting-poll-requests>
    </properties>
</channel-definition>

```

Note: You can also use secure AMF or HTTP channels in polling mode.

The caveat for using the `wait-interval-millis` is the utilization of available application server threads. Because this channel ties up one application server request handling thread for each parked poll request, this mechanism can have an impact on server resources and performance. Modern JVMs can typically support about 200 threads comfortably if given enough heap space. Check the maximum thread stack size (often 1 or 2 megabytes per thread) and make sure that you have enough memory and heap space for the number of application server threads you configure. This limitation does not apply to NIO-based endpoints.

To ensure that Flex clients using channels with `wait-interval-millis` do not lock up your application server, LiveCycle Data Services requires that you set the `max-waiting-poll-requests` property, which specifies the maximum number of waiting connections that LiveCycle Data Services should manage. This number must be set to a number smaller than the number of HTTP request threads your application server is configured to use. For example, you configure the application server to have at most 200 threads and allow at most 170 waiting poll requests. This setting would ensure that you have at least 30 application server threads to use for handling other HTTP requests. Your free application server threads should be large enough to maximize parallel opportunities for computation. Applications that are I/O heavy can require a large number of threads to ensure all I/O channels are utilized completely. Multiple threads are useful for the following operations:

Using different settings for these properties results in different behavior. For example, setting the `wait-interval-millis` property to 0 (zero) and setting the `polling-interval-millis` property to a nonzero positive value results in normal polling. Setting the `wait-interval-millis` property to a high value reduces the number of poll messages that the server must process, but the number of request handling threads on the server limits the total number of parked poll requests.

- Simultaneously writing responses to clients behind slow network connections
- Executing database queries or updates

- Performing computation on behalf of user requests

Another consideration for using `wait-interval-millis` is that LiveCycle Data Services must avoid monopolizing the available connections that the browser allocates for communicating with a server. The HTTP 1.1 specification recommends that browsers allocate at most two connections to the same server when the server supports HTTP 1.1. To avoid using more than one connection from a single browser, LiveCycle Data Services allows only one waiting thread for a given application server session at a time. If more than one Flash Player instance within the same browser process attempts to interact with the server using long polling, the server forces them to poll on the default interval with no server wait to avoid busy polling.

Streaming AMF and HTTP channels

The streaming AMF and HTTP channels are HTTP-based streaming channels that the LiveCycle Data Services server can use to push updates to clients using a technique called HTTP streaming. These channels give you the option of using standard HTTP for real-time messaging. This capability is supported for HTTP 1.1, but is not available for HTTP 1.0. There are also a number of proxy servers still in use that are not compliant with HTTP 1.1. When using a streaming channel, make sure that the channel has `connect-timeout-seconds` defined and the channel set has a channel to fall back to, such as an AMF polling channel.

LiveCycle Data Services gives you the additional option of using NIO-based endpoints with streaming AMF and HTTP channels. For more information about using NIO-based endpoints, see “[Configuring channels with NIO-based endpoints](#)” on page 55.

Using streaming AMF or HTTP channels/endpoints is like setting a long polling interval on a standard AMF or HTTP channel/endpoint, but the connection is never closed even after the server pushes the data to the client. By keeping a dedicated connection for server updates open, network latency is greatly reduced because the client and the server do not continuously open and close the connection. Unlike polling channels, because streaming channels keep a constant connection open, they can be adversely affected by HTTP connectors, proxies, reverse proxies or other network components that can buffer the response stream.

The following table describes the channel and endpoint configuration properties in the `services-config.xml` file that are specific to streaming AMF and HTTP channels/endpoints. The table includes the default property values as well as considerations for specific environments and applications.

Property	Description
<code>connect-timeout-seconds</code>	Using a streaming connection that passes through an HTTP 1.1 proxy server that incorrectly buffers the response sent back to the client hangs the connection. For this reason, you must set the <code>connect-timeout-seconds</code> property to a relatively short timeout period and specify a fallback channel such as an AMF polling channel.
<code>idle-timeout-minutes</code>	Optional channel property. Default value is 0. Specifies the number of minutes that a streaming channel is allowed to remain idle before it is closed. Setting the <code>idle-timeout-minutes</code> property to 0 disables the timeout completely, but it is a potential security concern.

Property	Description
max-streaming-clients	<p>Optional endpoint property. Default value is 10. Limits the number of Flex clients that can open a streaming connection to the endpoint. To determine an appropriate value, consider the number of threads available on your application server because each streaming connection open between a FlexClient and the streaming endpoints uses a thread on the server. Use a value that is lower than the maximum number of threads available on the application server.</p> <p>This value is for the number of Flex client application instances, which can each contain one or more MessageAgents (Producer or Consumer components).</p>
server-to-client-heartbeat-millis	<p>Optional endpoint property. Default value is 5000. Number of milliseconds that the server waits before writing a single byte to the streaming connection to make sure that the client is still available. This is important to determine when a client is no longer available so that its resources associated with the streaming connection can be cleaned up. A non-positive value disables this functionality.</p> <p>Note that this functionality does not keep the session alive. To keep a session alive, add a heartbeat-interval-millis property to the flex-client settings in the services-config.xml file. For more information, see “Configuration elements” on page 33.</p>

Property	Description
user-agent-settings	<p>Optional endpoint property. Use user agents to customize a long-polling or streaming endpoints for specific browsers. Long-polling and streaming endpoints require persistent HTTP connections which are limited differently by different browsers per session. A single long-poll connection requires two browser HTTP connections in order to send data in both directions: one for the streamed response from the server to the client that the channel hangs on to, and a second transient connection, drawn from the browser pool only when data needs to be sent to the server and this second transient connection is then immediately released back to the browser's connection pool. For client applications to function properly, the number of HTTP connections made by all clients running in the same session must stay within the limit established by the browser.</p> <p>The <code>max-persistent-connections-per-session</code> setting lets you limit the number of long-polling or streaming connections that can be made from clients in the same browser session. The <code>kickstart-bytes</code> setting indicates a certain number of bytes must be written before the endpoint can reliably use a streaming connection.</p> <p>There is a browser-specific limit to the number of connections allowed per session. By default, LiveCycle Data Services uses 1 as the value for <code>max-streaming-connections-per-session</code>. You can add browser-specific limits by specifying <code>user-agent</code> elements with a <code>match-on</code> value for specific browser user agents.</p> <p>The special match string "*" defines a default to be used if no string matches. The match strings "MSIE" and "Firefox" are always defined and must be specified explicitly to override the default settings. Specifying "*" does not do so. If you are using streaming channels, you should configure both the streaming and polling channels to the same values.</p> <p>The following example shows default user agent values:</p> <pre> <user-agent-settings> <!-- MSIE 5, 6, 7 limit is 2. <user-agent match-on="MSIE" max-persistent-connections- per-session="1" kickstart-bytes="2048"/> --> <!-- MSIE 8 limit is 6. <user-agent match-on="MSIE 8" max-persistent- connections-per-session="5" kickstart-bytes="2048"/> --> <!-- Firefox 1, 2 limit is 2. <user-agent match-on="Firefox" max-persistent- connections-per-session="1"/> --> <!-- Firefox 3 limit is 6. <user-agent match-on="Firefox/3" max-persistent- connections-per-session="5"/> --> <!-- Safari 3, 4 limit is 4. <user-agent match-on="Safari" max-persistent- connections-per-session="3"/> --> <!-- Chrome 0, 1, 2 limit is 6. <user-agent match-on="Chrome" max-persistent- connections-per-session="5"/> --> <!-- Opera 7, 9 limit is 4. <user-agent match-on="Opera" max-persistent- connections-per-session="3"/> --> <!-- Opera 8 limit is 8. <user-agent match-on="Opera 8" max-persistent- connections-per-session="7"/> --> <!-- Opera 10 limit is 8. <user-agent match-on="Opera/9.8" max-persistent- connections-per-session="7"/> --> </user-agent-settings></pre>

The following example shows streaming AMF and HTTP channel definitions:

```
<!-- AMF with streaming -->
<channel-definition id="my-amf-stream"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>

<!-- Secure AMF with streaming -->
<channel-definition id="my-secure-amf-stream"
    class="mx.messaging.channels.SecureStreamingAMFChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/securestreamingamf"
        class="flex.messaging.endpoints.SecureStreamingAMFEndpoint"/>
</channel-definition>

<!-- HTTP with streaming -->
<channel-definition id="my-http-stream"
    class="mx.messaging.channels.StreamingHTTPChannel">
    <endpoint url="http://servername:2080/myapp/messagebroker/streaminghttp"
        class="flex.messaging.endpoints.StreamingHTTPEndpoint"/>
</channel-definition>

<!-- Secure HTTP with streaming -->
<channel-definition id="my-secure-http-stream"
    class="mx.messaging.channels.SecureStreamingHTTPChannel">
    <endpoint url="https://servername:2080/myapp/messagebroker/securestreaminghttp"
        class="flex.messaging.endpoints.SecureStreamingHTTPEndpoint"/>
</channel-definition>
```

Configuring channels with NIO-based endpoints

Note: *NIO-based endpoints are not available in BlazeDS.*

AMF and HTTP (AMFX) channels offer long-polling and streaming channel support over HTTP. They are a good alternative to simple polling and approximate real-time push. However, in LiveCycle Data Services 2.5 they relied on servlet-based AMF and HTTP endpoints on the server. The servlet API requires a request-handler thread for each long-polling or streaming connection. A thread is not available to service other requests while it is servicing a long-poll parked on the server, or while it is servicing a streaming HTTP response. Therefore, the servlet implementation does not scale well for a large number of long-polling or streaming connections.

NIO-based AMF and HTTP endpoints address this scalability limitation by providing identical transport functionality from the client perspective, while avoiding the requirement of using one thread for each long-polling or streaming connection. NIO-based endpoints use the Java NIO API to service a large number of client connections in a non-blocking, asynchronous fashion.

Note: *Like servlet-based endpoints, NIO-based AMF and HTTP endpoints return a session id to inject into the client channel URL in the first response that is returned from the server. This allows client-server interaction to work when the client has cookies disabled. NIO endpoints also return a stand-in representation of a javax.servlet.HttpServlet object even though these endpoints do not use the Servlet API in any way. This stand-in object supports a subset of the full HttpServletRequest API. For more information, see the flex.messaging.io.http.StandInHttpServletRequest class the Javadoc API documentation.*

Configuring a shared socket server for NIO-based endpoints

You can configure one or more socket servers for use with NIO-based endpoints. The only reason to define more than one socket server is if you want to use both secure and insecure NIO endpoints. An underlying server either supports secure (SSL or TLS) connections or insecure connections, not both.

Do not define more than one socket server unless you need both secure and insecure support in the same application. When you do define more than one socket server, you can specify which server the endpoint of a channel definition uses. You use the `server` element in the `services-config.xml` file to create the server definition. The `servers` element can contain one or more `server` elements, as the following example shows:

```
<servers>
    <server id="my-nio-server" class="flex.messaging.socketserver.SocketServer">
    </server>

    <server id="secure-nio-server" class="flex.messaging.socketserver.SocketServer">
        <properties>
            <keystore-file>d:\keystores\localhoststore</keystore-file>
            <keystore-file>{context.root}/WEB-INF/flex/localhost.keystore</keystore-file>
            <keystore-password>changeit</keystore-password>
            <alias>lh</alias>
        </properties>
    </server>
</servers>
```

NIO-based endpoints within a channel definition reference a `<server>` definition by using the `<server ref="..."/>` property, as the following example shows:

```
<channel-definition id="my-nio-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://localhost:2080/mynioamf"
              class="flex.messaging.endpoints.NIOAMFEndpoint"/>
    <server ref="my-nio-server"/>
</channel-definition>
```

Note: One reason to share a server definition among multiple endpoint definitions is that a single port cannot be bound by more than one server definition. For example, if your application must define both a regular NIOAMFEndpoint as well as a StreamingNIOAMFEndpoint, you want clients to be able to reach either endpoint over the same port so these endpoints must reference a common server. If you do not require a shared server, you can configure socket server properties inside channel definitions.

An NIO-based endpoint can implicitly define and use an internal server definition by omitting the `<server ref="..."/>` property, and instead including any server configuration properties directly within its `<properties>` element. For example, if you define only a single NIO-based AMF or HTTP endpoint, omit the `<server ref="..."/>` element and define any desired properties within the endpoint definition. However, if a `<server ref="..."/>` property is included in the channel definition, any server-related configuration properties in the endpoint definition are ignored and must instead be defined directly for the server itself.

If you configure more than one NIO-based AMF or HTTP endpoint that is not secure, they should all reference a shared unsecured `server` definition. If you define more than one secure NIO-based AMF or HTTP endpoint, they should all reference a secured shared `<server>` definition. This configuration lets all insecure endpoints share the same insecure port, and lets all the secure endpoints share a separate, secure port.

Note: A server definition accepts either standard or secure connections based on whether it is configured to use a keystore. A server definition cannot service both standard and secure connections concurrently.

The endpoint uses the `class` attribute to register the protocol-specific connection implementation classes with the server. For information on the NIO HTTP implementation classes, see “[Defining channels and NIO-based endpoints](#)” on page 57.

Secure server definition

A secure server definition requires a digital certificate, and contains child elements for specifying a keystore filename and password. You can create public and private key pairs and self-signed certificates with the Java keytool utility. The common name (CN) of self-signed certificates must be set to `localhost` or the IP address on which the RTMPS endpoint is available. For information about key and certificate creation and management, see the Java keytool documentation at <http://java.sun.com>.

Optionally, you can store a keystore password in a separate file, possibly on removable media. Specify only one `keystore-password` or `keystore-password-file`. The following example shows a `keystore-password-file` element:

```
<keystore-password-file>a:\password</keystore-password-file>
```

Compatibility with previous versions of LiveCycle Data Services

The NIO-based AMF and HTTP endpoints are functionally equivalent to the existing servlet-based AMF and HTTP endpoints as far as the client is concerned. You should be able to run your existing applications, using the existing configuration files, when migrating your application to LiveCycle Data Services 2.6.

NIO-based endpoints do not support the following configuration options:

- `max-waiting-poll-requests`

NIO-based polling endpoints do not have the threading limitation that the servlet API imposes, so they do not require nor support the `max-waiting-poll-requests` property.

- `max-streaming-clients`

NIO-based streaming endpoints do not have the threading limitation that the servlet API imposes, so they do not require nor support the `max-streaming-clients` property.

Defining channels and NIO-based endpoints

When defining NIO-based endpoints, you specify one of the following classes to define the protocol-specific connection implementation. There is also a secure version of each of these endpoints that transports data over a secure connection. The names of the secure endpoints all start with the text “Secure”; for example, `SecureRTMPEndpoint`.

- `RTMPEndpoint`
- `NIOAMFEndpoint`
- `StreamingNIOAMFEndpoint`
- `NIOHTTPEndpoint`
- `StreamingNIOHTTPEndpoint`

The NIO-based AMF and HTTP endpoints implement part of the `HttpServletRequest` interface to expose headers and cookies in requests to custom `LoginCommands` or for other uses. You can access the `HttpServletRequest` instance by using the `FlexContext.getHttpRequest()` method. These endpoints do not implement the `HttpServletResponse` interface; therefore, the `FlexContext.getHttpResponse()` method returns `null`.

Supported methods from the `HttpServletRequest` interface include the following:

- `getCookies()`
- `getDateHeader()`

- `getHeader()`
- `getHeaderNames()`
- `getHeaders()`
- `getIntHeader()`

Calling an unsupported method of the `HttpServletRequest` interface throws an `UnsupportedOperationException` exception.

Configuring RTMP endpoints

You use RTMP channels to connect to an RTMP endpoint that supports real-time messaging and data. You can also configure these channels for long polling.

RTMPT connections are HTTP connections from the client to the server over which RTMP data is tunneled. When a direct RTMP connection is unavailable, the standard and secure channels automatically attempt to use RTMPT and tunneled RTMPS connections, respectively, on the RTMP endpoint with no additional configuration required.

Note: *You cannot use `server.port` tokens in an RTMP endpoint configuration. You can use the `server.name` token, but not when using clustering. If you have a firewall in place, you must open the port that you assign to the RTMP endpoint to allow RTMP traffic.*

RTMP is a protocol that is primarily used to stream data, audio, and video over the Internet to Flash Player clients. RTMP maintains a persistent connection with an endpoint and allows real-time communication. You can make an RTMP connection in one of the following ways depending upon the network setup of the clients:

- Direct socket connection.
- Tunneled socket connection through an HTTP proxy server using the CONNECT method.
- Active, adaptive polling using the browser HTTP stack. This mechanism does place additional computational load on the RTMP server and so reduces the scalability of the deployment.

RTMP direct socket connection

RTMP in its most direct form uses a simple TCP socket connection from the client machine directly to the RTMP server.

RTMPT Tunnelling with HTTP Proxy Server Connect

This connection acts like a direct connection but uses the proxy server support for the CONNECT protocol if supported. If this connection is not supported, the server falls back to using HTTP requests with polling.

RTMPT Tunnelling with HTTP requests

In this mode, the client uses an adaptive polling mechanism to implement two-way communication between the browser and the server. This mechanism is not configurable and is designed to use at most one connection between the client and server at a time to ensure that Flash Player does not consume all of the connections the browser allows to one server.

Connecting RTMP from the web tier to the application server tier

Currently, all RTMP connections must be made from the browser, or an HTTP proxy server in the case of a CONNECT-based tunnel connection, directly to the RTMP server. Because the NIO server is embedded in the application server tier with access to the database, in some server environments it is not easy to expose it directly to web clients. There are a few ways to address this problem in your configuration:

- You can expose a firewall and load balancer to web clients that can implement a TCP pass-through mode to the LiveCycle Data Services server.
- You can place a proxy server on the web tier that supports the HTTP CONNECT protocol so it can proxy connections from the client to the RTMP server. This provides real-time connection support by passing the web tier. The proxy server configuration can provide control over which servers and ports can use this protocol so it is more secure than opening up the RTMP server directly to clients. For an RTMP CONNECT tunnel to work, the client browser must be configured to use a specific HTTP proxy server.
- You can place a reverse proxy server on the web tier which proxies simple HTTP requests to the NIO server. This approach only works in RTMPT polling mode.
- You can deploy an Edge Server in your DMZ. Clients connect to the Edge Server over RTMP, and the Edge Server then forwards the client requests to a LiveCycle Data Services server in the application tier. Because the Edge Server is running in the DMZ, you are not exposing the secure application tier of your internal network.

Configuring a standard RTMPChannel/RTMPEndpoint

Use a standard RTMPChannel/RTMPEndpoint only when transmitting data that does not require security. The following example shows an RTMP channel definition:

```
<channel-definition id="my-rtmp"
    class="mx.messaging.channels.RTMPChannel">
    <endpoint url="rtmp://{{server.name}}:2035"
        class="flex.messaging.endpoints.RTMPEndpoint"/>
    <properties>
        <idle-timeout-minutes>20</idle-timeout-minutes>
    </properties>
</channel-definition>
```

You can use the optional `rtmpt-poll-wait-millis-on-client` element when clients are connected over RTMPT while performing adaptive polling. After receiving a server response, clients wait for an absolute time interval before issuing the next poll request rather than using an exponential back-off up to a maximum poll wait time of 4 seconds. Allowed values are integers in the range of 0 to 4000, for example:

```
<rtmpt-poll-wait-millis-on-client>0</rtmpt-poll-wait-millis-on-client>
```

You can set the optional `block-rtmpt-polling-clients` element to `true` to allow the server to be configured to drop the inefficient polling connection. This action triggers the client to fall back to the next channel in its channel set. The default value is `false`.

When an RTMPChannel internally falls back to tunneling, it sends its HTTP requests to the same domain/port that is used for regular RTMP connections and these requests must be sent over a standard HTTP port so that they make it through external firewalls that some clients could be behind. For RTMP to work, you must bind your server-side RTMP endpoint for both regular RTMP connections and tunneled connections to port 80. When the client channel falls back to tunneling, it always goes to the domain/port specified in the channel endpoint URI.

If you have a switch or load balancer that supports virtual IPs, you can use a deployment as the following table shows, using a virtual IP (and no additional NIC on the LiveCycle Data Services server) where client requests first hit a load balancer that routes them back to a backing LiveCycle Data Services server:

Public Internet	Load balancer	LiveCycle Data Services
Browser client makes HTTP request	my.domain.com:80	Servlet container bound to port 80
Browser client uses RTMP/T	rtmp.domain.com:80 (virtual IP address)	RTMPEndpoint bound to any port; for example, 2037

The virtual IP/port in this example (rtmp.domain.com:80) is configured to route back to port 2037 or whatever port the RTMPEndpoint is configured to use on the LiveCycle Data Services server. In this approach, the RTMPEndpoint does not have to bind to a standard port because the load balancer publicly exposes the endpoint via a separate virtual IP address on a standard port.

If you do not have a switch or load balancer that supports virtual IPs, you need a second NIC to allow the RTMPEndpoint to bind port 80 on the same server as the servlet container that binds to port 80 on the primary NIC, as the following table shows:

Public Internet	LiveCycle Data Services NICs	LiveCycle Data Services
Browser client makes HTTP request	my.domain.com:80	Servlet container bound to port 80 on primary NIC using primary domain/IP address
Browser client uses RTMP/T	rtmp.domain.com:80	RTMPEndpoint bound to port 80 on a second NIC using a separate domain/IP address

In this approach the physical LiveCycle Data Services server has two NICs, each with its own static IP address. The servlet container binds port 80 on the primary NIC and the RTMPEndpoint binds port 80 on the secondary NIC. This allows traffic to and from the server to happen over a standard port (80) regardless of the protocol (HTTP or RTMP). The key take away is that for the RTMPT fallback to work smoothly, the RTMPEndpoint must be exposed to Flex clients over port 80, which is the standard HTTP port that generally allows traffic (unlike nonstandard RTMP ports like 2037, which are usually closed by IT departments).

To bind the RTMPEndpoint to port 80, one of the approaches above is required. For secure RTMPS (direct or tunneled) connections, the information above is the same but you would use port 443. If an application requires both secure and non-secure RTMP connections with tunneling fallback, you do not need three NICs; you need only two. The first NIC services regular HTTP/HTTPS traffic on the primary IP address at ports 80 and 443, and the second NIC services RTMP and RTMPS connections on the secondary IP at ports 80 and 443. If a load balancer is in use, it is the same as the first option above where LiveCycle Data Services has a single NIC and the different IP/port combinations are defined at the load balancer.

Configuring NIO-based AMF and HTTP endpoints

The NIO-based AMF and HTTP endpoint classes support simple polling, long-polling, and streaming requests. Like the socket server configuration options, these endpoints support the existing configuration options defined for the RTMPEndpoint, AMFEndpoint and HTTPEndpoint classes, and the following additional configuration options:

Option	Description
session-cookie-name	(Optional) Default value is <code>AMFSessionId</code> . The name used for the session cookie set by the endpoint. If you define multiple NIO-based endpoints and specify a custom cookie name, it must be the same for all NIO-based AMF and HTTP endpoints.
session-timeout-minutes	(Optional) Default value is 15 minutes. The session timeout for inactive NIO HTTP sessions, in minutes. Long-poll requests or streaming connections that uses <code>server-to-client-heartbeat-millis</code> keep the session alive, as will any request from the client. If you define multiple NIO-based endpoint and specify a custom session timeout, it must be the same for all NIO-based AMF and HTTP endpoints.
max-request-line-size	(Optional) Default value is 8192 characters. Requests that contain request or header lines that exceed this value are dropped and the connection is closed.
max-request-body-size	(Optional) Default is 5 MB. The maximum POST body size, in bytes, that the endpoint accepts. Requests that exceed this limit are dropped and the connection is closed.

Channels that use NIO-based endpoints

The following channel definition defines an RTMP channel for use with RTMP endpoints.

```
<!-- NIO RTMP -->
<channel-definition id="my-rtmp" class="mx.messaging.channels.RTMPChannel">
    <endpoint url="rtmp://servername:2038"
        class="flex.messaging.endpoints.RTMPEndpoint"/>
    <properties>
        <idle-timeout-minutes>0</idle-timeout-minutes>
        <bind-address>10.132.64.63</bind-address>
        <bind-port>2035</bind-port>
    </properties>
</channel-definition>
```

The following channel definitions define AMF channels for use with NIO-based AMF endpoints.

```
<!-- NIO RTMP -->
<channel-definition id="my-rtmp" class="mx.messaging.channels.RTMPChannel">
    <endpoint url="rtmp://servername:2038"
        class="flex.messaging.endpoints.RTMPEndpoint"/>
    <properties>
        <idle-timeout-minutes>0</idle-timeout-minutes>
        <bind-address>10.132.64.63</bind-address>
        <bind-port>2035</bind-port>
    </properties>
</channel-definition>

<!-- NIO AMF -->
<channel-definition id="my-nio-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://servername:2080/nioamf"
        class="flex.messaging.endpoints.NIOAMFEndpoint"/>
    <server ref="my-nio-server"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>
```

```

<!-- NIO AMF with polling -->
<channel-definition id="my-nio-amf-poll" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://servername:2080/nioamfpoll"
        class="flex.messaging.endpoints.NIOAMFEndpoint"/>
    <server ref="my-nio-server"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>3000</polling-interval-millis>
    </properties>
</channel-definition>

<!-- NIO AMF with long polling -->
<channel-definition id="my-nio-amf-longpoll" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://servername:2080/nioamflongpoll"
        class="flex.messaging.endpoints.NIOAMFEndpoint"/>
    <server ref="my-nio-server"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>0</polling-interval-millis>
        <wait-interval-millis>-1</wait-interval-millis>
    </properties>
</channel-definition>

<!-- NIO AMF with streaming -->
<channel-definition id="my-nio-amf-stream"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint url="http://servername:2080/nioamfstream"
        class="flex.messaging.endpoints.StreamingNIOAMFEndpoint"/>
    <server ref="my-nio-server"/>
    <properties>
        <server-to-client-heartbeat-millis>10000</server-to-client-heartbeat-millis>
        <user-agent-settings>
            <user-agent match-on="MSIE"
                kickstart-bytes="2048" max-streaming-connections-per-session="1"/>
            <user-agent match-on="Firefox"
                kickstart-bytes="0" max-streaming-connections-per-session="1"/>
        </user-agent-settings>
    </properties>
</channel-definition>

<!-- Secure NIO AMF -->
<channel-definition id="secure-nio-amf" class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://servername:2443/securenioamf"
        class="flex.messaging.endpoints.SecureNIOAMFEndpoint"/>
    <server ref="secure-nio-server"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>
```

The following channel definitions define HTTP channels for use with NIO-based HTTP endpoints:

```

<!-- NIO HTTP -->
<channel-definition id="my-nio-http" class="mx.messaging.channels.HTTPChannel">
    <endpoint url="http://servername:2080/niohttp"
        class="flex.messaging.endpoints.NIOHTTPEndpoint"/>
    <server ref="my-nio-server"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>

<!-- NIO HTTP with streaming -->
<channel-definition id="my-nio-http-stream"
    class="mx.messaging.channels.StreamingHTTPChannel">
    <endpoint url="http://servername:2080/niohttpstream"
        class="flex.messaging.endpoints.StreamingNIOHTTPEndpoint"/>
    <server ref="my-nio-server"/>
</channel-definition>

<!-- Secure NIO HTTP -->
<channel-definition id="secure-nio-http" class="mx.messaging.channels.SecureHTTPChannel">
    <endpoint url="https://servername:2443/secureniohttp"
        class="flex.messaging.endpoints.SecureNIOHTTPEndpoint"/>
    <server ref="secure-nio-server"/>
    <properties>
        <polling-enabled>false</polling-enabled>
    </properties>
</channel-definition>

```

Monitoring an NIO-based endpoint through a proxy service

You can use a proxy service to monitor traffic to and from an NIO-based endpoint. To do so, set the `url` property of the `endpoint` element in a channel definition to use the port on which you want the proxy service to listen. The client uses the `url` property of the `endpoint` element. Set the `bind-port` property of the NIO server to the port that the server endpoint should listen on.

For the following channel definition, you would configure your proxy service to listen for requests on port 9999 and forward requests to port 2081:

```

<channel-definition
    class="mx.messaging.channels.StreamingAMFChannel" id="qa-nio-amf-stream-proxy">
    <endpoint class="flex.messaging.endpoints.StreamingNIOAMFEndpoint"
        url="http://yourserver:9999/nioamfstreamproxy"/>
    <server ref="nio-server"/>
    <properties>
        <bind-port>2081</bind-port>
        <server-to-client-heartbeat-millis>5000</server-to-client-heartbeat-millis>
    </properties>
</channel-definition>

```

Configuring the socket server

You configure one or more socket server definitions in the `services-config.xml` file. Once defined, you can specify which server a channel definition uses.

Note: You can optionally set any of the socket server properties directly in a channel definition for situations where you are not sharing the same socket server for more than one endpoint.

Use a secure channel definition to connect to the endpoint over Transport Layer Security (TLS) for RTMP-based endpoints or SSL for servlet-based endpoints. When using RTMP, the channel falls back to tunneled RTMPS when a direct connection is unavailable.

The following table describes the configuration options for the socket server:

Option	Description
Security Related Options	
max-connection-count	(Optional) The total number of open connections that the server accepts. Clients that attempt to connect beyond this limit are rejected. Setting <code>max-connection-count</code> to 0, the default, disables this limitation. Note that this is a potential security concern.
connection-idle-timeout-minutes	(Optional) The number of minutes that a streaming channel is allowed to remain idle before it is closed. Setting <code>idle-timeout-minutes</code> to 0, the default value, disables the timeout. Note that this is a potential security concern.
idle-timeout-minutes	Supported but deprecated. See <code>connection-idle-timeout-minutes</code> above.
whitelist	(Optional) Contains client IP addresses that are permitted to connect to the server. When defined, client IPs must satisfy this filter to connect. The <code>blacklist</code> option takes precedence over the <code>whitelist</code> option if the client IP is a member of both sets. Define a <code>whitelist</code> by creating a list of <code>ip-address</code> and <code>ip-address-pattern</code> elements. The <code>ipaddress</code> element supports simple IP matching, allowing you to use a wildcard symbol, *, for individual bytes in the address. The <code>ip-address-pattern</code> element supports regular-expression pattern matching of IP addresses to support range-based IP filtering, as the following example shows: <pre><whitelist> <ip-address>10.132.64.63</ip-address> <ip-address-pattern>240.*</ip-address-pattern> </whitelist></pre>
blacklist	(Optional) Contains client IPs that are restricted from accessing the server. The <code>blacklist</code> option takes precedence over the <code>whitelist</code> option if the client IP is a member of both sets. Blacklists take a list of <code>ip-address</code> and <code>ip-address-pattern</code> elements, as the following example shows: <pre><blacklist> <ip-address>10.132.64.63</ip-address> <ip-address-pattern>240.63.*</ip-address-pattern> </blacklist></pre> For more information on the <code>ip-address</code> and <code>ip-address-pattern</code> elements, see the <code>whitelist</code> option.
cross-domain-path	Path to a cross domain policy file. For more information, see “ Using cross domain policy files with the NIO server ” on page 68.

Option	Description
enabled-cipher-suites	<p>A cipher suite is a set of encryption algorithms. Secure NIO socket server definitions or secure NIO (RTMP, AMF, or HTTP) endpoint definitions can define a restricted set of cipher suites to use with secure connections, as the following examples shows:</p> <pre data-bbox="644 440 1199 565"><enabled-cipher-suites> <cipher-suite>SSL_RSA_WITH_128_MD5</cipher-suite> <cipher-suite>TLS_DHE_DSS_WITH_AES_128_CBC_SHA</cipher-suite> </enabled-cipher-suites></pre> <p>If left undefined, the default set of cipher suites for the security provider implementation of the Java virtual machine (JVM) are used for secure connection handshaking. If an unsupported cipher suite is defined, an error occurs and a ConfigurationException exception is thrown when the server starts.</p>
General Server Options	
bind-address	(Optional) Binds the server to a specific network interface. If left unspecified, the server binds to all local network interfaces on startup.
bind-port	(Optional) Binds the server to a port that differs from the public port that clients connect to. This allows clients to connect to a public port on a firewall or load balancer that forwards traffic to this internal port.
reuse-address-enabled	(Optional) When set to false disables address reuse by the socket server. This causes the SocketServer to not start when it cannot bind to a port because the port is in a TIME_WAIT state. The default value is true, which allows the socket server to start if the port is in a TIME_WAIT state.
Connection Configuration Options	
connection-buffer-type	<p>(Optional) Controls the type of byte buffers that connections use to read and write from the network. Supported values include:</p> <ul data-bbox="644 1248 1150 1305" style="list-style-type: none"> • heap (default) specifies to use heap-allocated ByteBuffers. • direct specifies to use DirectByteBuffers. <p>Performance varies depending upon JRE and operating system.</p>
connection-read-buffer-size	(Optional) Default value is 8192 (8 KBytes). Size, in bytes, for the read buffer for connections.
connection-write-buffer-size	(Optional) Default value is 8192 (8 KBytes). Size, in bytes, for the write buffer for connections.
Control the threading behavior of the server in the absence of a WorkManager.	
websphere-workmanager-jndi-name	(Optional) For IBM WebSphere deployment, the WorkManager configured and referenced as the worker thread pool for the server. A WorkManager must be referenced to support JTA, which the Data Management Service uses.
max-worker-threads	(Optional) The upper bound on the number of worker threads that the server uses to service connections. Defaults to unbounded with idle threads being reaped based upon the idle-worker-thread-timeout-millis value. Ignored if the server is using a WorkManager.

Option	Description
min-worker-threads	(Optional) A nonzero, lower bound on the number of worker threads that the server uses to service connections. Defaults to 0. If nonzero, on server start up this number of worker threads is prestarted. Ignored if the server is using a WorkManager.
idle-worker-thread-timeout-millis	(Optional) Time, in minutes, that a worker thread must be idle before it is released. Defaults to 1. Ignored if the server is using a WorkManager.
worker-thread-priority	(Optional) Desired priority for the server threads. Defaults to Thread.NORM_PRIORITY. Ignored if the server is using a WorkManager.
Accept or and Reactor Configuration Options	
accept-backlog	(Optional) The suggested size of the accept queue for the server socket to buffer new connect requests when the server is too busy to handshake immediately. Default is the platform default value for a server socket.
accept-thread-priority	(Optional) Allows the priority for accepting new connections to be adjusted either above or below the general server priority for servicing existing connections. Default is the thread priority of the worker thread pool or the WorkManager (whichever is used).
reactor-count	(Optional) Splits IO read/write readiness selection for all existing connections to the server out across all available processors and divides the total readiness selection set by the number of reactors to avoid having a single Selector manage the state for all connections. Default is the number of available physical processors on the machine.
TLS/SSL Configuration Options	
keystore-type	(Optional) Default value is JKS. Specifies the keystore type. The supported values include: JKS and PKCS12.
keystore-file	Points to the actual keystore file on the running server. The path to the file may be specified as an absolute path, as in: <keystore-file>C:\keystore.jks</keystore-file> Or as a relative to the WEB-APP directory using the {context.root} token, as in: <keystore-file>{context.root}/WEB-INF/certs/keystore.jks</keystore-file>
keystore-password	The keystore password.
keystore-password-file	Path to a text file on external or removable media that contains the password value. Ignored if keystore-password is specified.
keystore-password-obfuscated	(Optional) Default value is false. Set to true if an obfuscated value is stored in the configuration file and a password deobfuscator has been registered for use.

Option	Description
password-deobfuscator	<p>(Optional) The class name of a password deobfuscator used to handle obfuscated passwords safely at run time.</p> <p>A password deobfuscator implements the <code>flex.messaging.util.PasswordDeobfuscator</code> interface, which contains a <code>deobfuscate()</code> and a <code>destroy()</code> method. The <code>flex.messaging.util.AbstractPasswordDeobfuscator</code> class is an abstract class that implements the <code>destroy()</code> method. Your deobfuscator class can extend this class, and just implement a <code>deobfuscate()</code> method. The <code>deobfuscate()</code> method takes a string that contains the obfuscated password and returns a <code>char[]</code> containing the deobfuscated password.</p> <p>For more information, see the JavaDoc for the <code>flex.messaging.util.PasswordDeobfuscator</code> interface and the <code>flex.messaging.util.AbstractPasswordDeobfuscator</code> abstract base class.</p>
algorithm	<p>(Optional) Only configure this option when using a third-party security provider implementation. Defaults to the current default JVM algorithm. The default JVM algorithm can be controlled by setting the <code>ssl.KeyManagerFactory.algorithm</code> security property.</p> <p>Example usage:</p> <ul style="list-style-type: none"> • <code><algorithm>Default</algorithm></code> - Explicitly uses the default JVM algorithm. • <code><algorithm>SunX509</algorithm></code> - Uses Sun algorithm; requires the Sun security provider. • <code><algorithm>IbmX509</algorithm></code> - Uses IBM algorithm; requires the IBM security provider.
alias	(Optional) If a keystore is used that contains more than one server certificate, use this property to define which certificate to use to authenticate the server end of SSL/TLS connections.
protocol	(Optional) Default value is TLS. The preferred protocol for secure connections.
Socket Configuration Options (for accepted sockets)	
socket-keepalive-enabled	(Optional) Enables/Disables <code>SO_KEEPALIVE</code> for sockets the server uses.
socket-oobinline-enabled	(Optional) Enables/Disables <code>OOINLINE</code> (receipt of TCP urgent data). Defaults to <code>false</code> , which is the JVM default.
socket-receive-buffer-size	(Optional) Sets the <code>SO_RCVBUF</code> option for sockets to this value, which is a hint for the size of the underlying network I/O buffer to use.
socket-send-buffer-size	(Optional) Sets the <code>SO_SNDBUF</code> option for sockets to this value, which is a hint for the size of the underlying network I/O buffer to use.
socket-linger-seconds	(Optional) Enables/Disables <code>SO_LINGER</code> for sockets the server uses where the value indicates the linger time for a closing socket in seconds. A value of -1 disables socket linger.
socket-tcp-no-delay-enabled	(Optional) Enables/Disables <code>TCP_NODELAY</code> , which enables or disables the use of the Nagle algorithm by the sockets.
socket-traffic-class	(Optional) Defines the traffic class for the socket.

Using cross domain policy files with the NIO server

By default, the Flash Player security sandbox only allows a SWF file to make requests back to the same domain that the SWF file originated from. You use a cross domain policy file to deal with this security restriction. If a SWF file makes a request to a domain that is different than the one from which it originated, Flash Player makes a request to the new domain for a cross domain policy file that can provide the SWF file with additional permissions to communicate with the domain.

For servlet-based endpoints, LiveCycle Data Services depends on the application server for cross domain policy file handling. For NIO-based AMF and HTTP endpoints, LiveCycle Data Services must depend on the NIO server for cross domain policy file handling and not the application server itself. This is because the NIO server listens for requests on a different port than the application server.

If your application makes a request to an AMF or HTTP endpoint at the `http://domainA:8400/lcds/messagebroker/amfpolling` and domainA is not the same domain the SWF file was downloaded from, Flash Player makes a cross domain policy file request to `http://domainA:8400`. To allow the application to make the request to the AMF endpoint, you must place a cross domain policy file in the root directory of your application server. For example, with a typical installation of Tomcat, this would be the `TOMCAT_HOME\webapps\ROOT` directory.

If your application makes a request to an NIO-based AMF or HTTP endpoint at the `http://domainA:1234/nioamfpolling` and domainA is not the same domain the SWF file was downloaded from, Flash Player makes a cross domain policy file request to `http://domainA:1234`. Since this request goes directly to the NIO server and not to the application server, you must configure the NIO server to serve the cross domain policy file. To do this, you add a `cross-domain-path` property to your NIO server settings in your `services-config.xml` file as the following example shows:

```
<server id="my-nio-server" class="flex.messaging.SocketServer">
  <properties>
    <http>
      <cross-domain-path>crossdomain.xml</cross-domain-path>
    </http>
  </properties>
</server>
```

Because this setting only applies to NIO-based AMF and HTTP endpoints it goes in the http section of configuration settings. The NIO server expects cross domain policy files to be under the WEB-INF/flex directory of the LiveCycle Data Services web application. In the previous example, you would place a file called `crossdomain.xml` with your cross domain policy settings in the WEB-INF/flex directory of your LiveCycle Data Services web application.

You can also place cross domain files in subdirectories under WEB-INF/flex. This lets you have cross domain policy files for individual NIO-based endpoints. For example, if you want a cross domain policy file that only applies to your NIO-based AMF endpoint at `http://domainA:1234/nioamfpolling`, you would configure your application as follows:

- Use a `cross-domain-path` of `/nioamfpolling`, which causes the NIO server to search for a cross domain policy file under `/WEB-INF/flex/nioamfpolling`.
- The cross domain path must match the cross domain policy request. This means that to use the cross domain policy file under `/WEB-INF/flex/nioamfpolling`, the request must be made to `http://domainA:1234/nioamfpolling/crossdomain.xml`.
- By default, the cross domain policy request is made to the root of the application; for example, `http://domainA:1234/crossdomain.xml`. To override this behavior in your application, you must use the `Security.loadPolicyFile()` method; in your application, before you make the request to the endpoint, you would add the following line of code.

```
Security.loadPolicyFile("http://domainA:1234/nioamfpolling/crossdomain.xml");
```

This causes Flash Player to attempt to load a cross domain policy file with the supplied URL. In this case, the NIO server returns a cross domain policy file from the WEB-INF/flex/nioamfpolling directory and the permissions in the policy file are applied to requests to the http://domainA:1234/nioamfpolling endpoint only.

Channel and endpoint recommendations

The NIO-based AMF and HTTP endpoints use the same client-side channels as their servlet-based endpoint counterparts. The main advantage that NIO-based endpoints have over servlet-based endpoints is that they scale better. If a web application is not servicing general servlet requests, you can configure the servlet container to bind non-standard HTTP and HTTPS ports, leaving ports 80 and 443 free for your NIO-based endpoints. You still have access to the servlet-based endpoints if you want to use them instead.

Note: The LiveCycle Data Services installation includes an NIO load testing tool with sample test driver classes and a readme file in the install_root/resources/load-testing-tool directory. You can use this tool in a Java client application to load test LiveCycle Data Services NIO-based channels/endpoints running in a LiveCycle Data Services web application.

Use the servlet-based endpoints when you must include third-party servlet filter processing of requests and responses or when you must access data structures in the application server HttpSession. NIO -based AMF and HTTP endpoints are not part of the servlet pipeline, so, although they provide a FlexSession in the same manner that RTMP connections do, these session instances are disjoint from the J2EE HttpSession.

If you are only using remote procedure calls or using the Data Management Service without auto synchronization, you can use the AMFChannel.

Note: The HTTPChannel is the same as the AMFChannel behaviorally, but serializes data in an XML format called AMFX. This channel only exists for customers who require all data sent over the wire to be non-binary for auditing purposes. There is no other reason to use this channel instead of the AMFChannel for RPC-based applications.

The process for using real-time data push to web clients is not as simple as the RPC scenario. There are a variety of trade-offs, and benefits and disadvantages to consider. Although the answer is not simple, it is prescriptive based on the requirements of your application.

If your application uses both real-time data push as well as RPC, you do not need to use separate channels. All of the channels listed can send RPC invocations to the server. Use a single channel set, possibly containing just a single channel, for all of your RPC, messaging and data management components.

Servlet-based endpoints

Some servlet-based channel/endpoint combinations are preferred over others, depending on your application environment. Each combination is listed here in order of preference. Although servlet-based channels/endpoints are listed first, the equivalent NIO-based versions are preferred when NIO is available and acceptable in your application environment.

1. AMFChannel/Endpoint configured for long polling (no fallback needed)

The channel issues polls to the server in the same way as simple polling, but if no data is available to return immediately the server parks the poll request until data arrives for the client or the configured server wait interval elapses.

The client can be configured to issue its next poll immediately following a poll response making this channel configuration feel like real-time communication.

A reasonable server wait time would be one minute. This eliminates the majority of busy polling from clients without being so long that you're keeping server sessions alive indefinitely or running the risk of a network component between the client and server timing out the connection.

Benefits	Disadvantages
Valid HTTP request/response pattern over standard ports that nothing in the network path will have trouble with.	When many messages are being pushed to the client, this configuration has the overhead of a poll round trip for every pushed message or small batch of messages queued between polls. Most applications are not pushing data so frequently for this to be a problem.
	The Servlet API uses blocking IO, so you must define an upper bound for the number of long poll requests parked on the server at any single instant. If your number of clients exceeds this limit, the excess clients devolve to simple polling on the default 3-second interval with no server wait. For example, if your server request handler thread pool has a size of 500, you could set the upper bound for waited polls to 250, 300, or 400 depending on the relative amount of non-poll requests you expect to service concurrently.

2. StreamingAMFChannel/Endpoint (in a channel set followed by the polling AMFChannel for fallback)

Because HTTP connections are not duplex, this channel sends a request to open an HTTP connection between the server and client, over which the server writes an infinite response of pushed messages. This channel uses a separate transient connection from the browser connection pool for each send it issues to the server. The streaming connection is used purely for messages pushed from the server down to the client. Each message is pushed as an HTTP response chunk (HTTP 1.1 Transfer-Encoding: chunked).

Benefits	Disadvantages
No polling overhead associated with pushing messages to the client.	Holding onto the open request on the server and writing an infinite response is not typical HTTP behavior. HTTP proxies that buffer responses before forwarding them can effectively consume the stream. Assign the channel's 'connect-timeout-seconds' property a value of 2 or 3 to detect this and trigger fallback to the next channel in your channel set.
Uses standard HTTP ports so firewalls do not interfere and all requests/responses are HTTP so packet inspecting proxies won't drop the packets.	No support for HTTP 1.0 client. If the client is 1.0, the open request is faulted and the client falls back to the next channel in its channel set.
	The Servlet API uses blocking IO so as with long polling above, you must set a configured upper bound on the number of streaming connections you allow. Clients that exceed this limit are not able to open a streaming connection and will fall back to the next channel in their channel set.

3. AMFChannel/Endpoint with simple polling and piggybacking enabled (no fallback needed)

This configuration is the same as simple polling support but with piggybacking enabled. When the client sends a message to the server between its regularly scheduled poll requests, the channel piggybacks a poll request along with the message being sent, and the server piggybacks any pending messages for the client along with the response.

Benefits	Disadvantages
Valid HTTP request/response pattern over standard ports that nothing in the network path will have trouble with.	Less real-time behavior than long polling or streaming. Requires client interaction with the server to receive pushed data faster than the channel's configured polling interval.
User experience feels more real-time than with simple polling on an interval.	
Does not have thread resource constraints like long polling and streaming due to the blocking IO of the Servlet API.	

NIO-based endpoints

Some NIO-based channel/endpoint combinations are preferred over others, depending on your application environment. Each combination is listed here in order of preference.

1. RTMPChannel/Endpoint (in a channel set with fallback to NIO AMFChannel configured to long poll)

The RTMPChannel creates a single duplex socket connection to the server and gives the server the best notification of Flash Player being shut down. If the direct connect attempt fails, the player attempts a CONNECT tunnel through an HTTP proxy if one is defined by the browser, resulting in a direct, tunneled duplex socket connection to the server. In the worst case, the channel falls back to adaptive HTTP requests that tunnel RTMP data back and forth between client and server, or it fails to connect entirely.

Benefits	Disadvantages
Single, stateful duplex socket that gives clean, immediate notification when a client is closed. The NIO-based AMF and HTTP channels/endpoints generally do not receive notification of a client going away until the HTTP session on the server times out. That is not optimal for a call center application where you must know whether representatives are online or not.	RTMP generally uses a non-standard port so it is often blocked by client firewalls. Network components that use stateful packet inspection might also drop RTMP packets, killing the connection. Fallback to HTTP CONNECT through a proxy or adaptive HTTP tunnel requests is difficult in deployment scenarios within a Java servlet container which generally already has the standard HTTP ports bound. This requires a non-trivial networking configuration to route these requests to the RTMPEndpoint. By using an Edge Server in the DMZ to forward requests to a LiveCycle Data Services server in the application tier, you can avoid or work around some of these issues. For more information about the Edge Server see " Edge Server " on page 391.
The Flash Player internal fallback to HTTP CONNECT to traverse an HTTP proxy if one is configured in the browser gives the same benefit as above, and is a technique that is not possible from ActionScript or JavaScript.	

2. NIO AMFChannel/Endpoint configured for long polling (no fallback needed)

This configuration is behaviorally the same as the servlet-based AMFChannel/AMFEndpoint but uses an NIO server and minimal HTTP stack to support scaling up to thousands of connections.

Benefits	Disadvantages
The same benefits as mentioned for RTMPChannel/Endpoint, along with much better scalability and no configured upper bound on the number of parked poll requests.	Because the servlet pipeline is not used, this endpoint requires more network configuration to route requests to it on a standard HTTP port if you must concurrently service HTTP servlet requests. However, it can share the same port as any other NIO-based AMF or HTTP endpoint for the application.

3. NIO StreamingAMFChannel/Endpoint (in a channel set followed by the polling AMFChannel for fallback)

This configuration is behaviorally the same as the servlet-based StreamingAMFChannel/Endpoint but uses an NIO server and minimal HTTP stack to support scaling up to thousands of connections.

Benefits	Disadvantages
The same benefits as mentioned for NIO AMFChannel/Endpoint configured for Long Polling, along with much better scalability and no configured upper bound on the number of streaming connections.	Because the servlet pipeline is not being used, this endpoint requires more network configuration to route requests to it on a standard HTTP port. It must concurrently service HTTP servlet requests. However, it can share the same port as any other NIO-based AMF or HTTP endpoint for the application. Unlike RTMP, which provides a full duplex socket connection, this scenario requires one connection for regular HTTP requests and responses, and another connection for server-to-client updates.

4. NIO AMFChannel/Endpoint with simple polling enabled (no long polling) and piggybacking enabled (no fallback needed)

Behaviorally the same as the equivalent servlet-based configuration, but uses an NIO server and minimal HTTP stack to support scaling up to thousands of connections.

Benefits	Disadvantages
Same benefits as the servlet-based piggybacking configuration. Shares the same FlexSession as other NIO-based AMF and HTTP endpoints	Same disadvantages as the servlet-based piggybacking configuration. Because the servlet pipeline is not being used, this endpoint requires more network configuration to route requests to it on a standard HTTP port if you need to concurrently service HTTP servlet requests. However, it can share the same port as any other NIO-based AMF or HTTP endpoint for the application.

Using LiveCycle Data Services clients and servers behind a firewall

Because servlet-based endpoints use standard HTTP requests, communicating with clients inside firewalls usually works, as long as the client-side firewall has the necessary ports open. Using the standard HTTP port 80 and HTTPS port 443 is recommended because many firewalls block outbound traffic over non-standard ports.

When working with NIO-based endpoints, using AMF and HTTP endpoints instead of RTMP endpoints when clients are inside a firewall allows the clients to connect to the server through their client-side firewall without running into a blocked port. This method also allows packets to traverse stateful proxies that drop packets that they do not recognize, but this requires that you expose the NIO-based AMF and HTTP endpoints on the standard HTTP port 80 and the standard HTTPS port 443.

You can use load balancers with the NIO-based AMF and HTTP endpoints. You must configure the load balancer to do sticky sessions based on the session cookie `AMFSessionId`, or based upon the session id embedded in request URLs that match the format of J2EE-encoded URLs but use an `AMFSessionId` token rather than the J2EE `jsessionid` token. For RTMP endpoints, you should apply TCP round-robin connectivity to the load-balanced RTMP port.

You must configure server-side firewalls the same way you would to allow access to servlet-based AMF and HTTP endpoints. Be aware of the following potential issues when using web servers/connectors in front of NIO-based AMF and HTTP endpoints:

- The web server/connector could buffer streamed chunks in the response, interfering with real-time streaming. This is not an issue for polling or long-polling, where a complete HTTP response is returned whenever data is sent to the client.
- The web server/connector may not use asynchronous IO, in which case a single webserver or connector most likely will not scale up to thousands of persistent concurrent HTTP connections.

The protocols that the various client channels use are hard coded. For example, the AMFChannel always uses HTTP, while the SecureAMFChannel always uses HTTPS. One thing to watch for when using a SecureAMFChannel/SecureAMFEndpoint combination is an issue with Internet Explorer related to no-cache response headers and HTTPS. By default, no-cache response headers are enabled on HTTP-based endpoints. This causes problems for Internet Explorer browsers. You can suppress these response headers by adding the following configuration property to your endpoint:

```
<add-no-cache-headers>false</add-no-cache-headers>
```

When you have a firewall/reverse HTTP proxy in your deployment that handles SSL for you, you must mix and match your channel and endpoint. You need the client to use a secure channel and the server to use an insecure endpoint, as the following example shows:

```
<channel-definition id="secure-amf" class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://<<firewall ip:port>>/{context.root}/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <add-no-cache-headers>false</add-no-cache-headers>
    ...

```

The channel class uses HTTPS to hit the firewall/proxy, and the endpoint URL must point at the firewall/proxy. Because SSL is handled in the middle, you want the endpoint class used by LiveCycle Data Services to be the insecure AMFEndpoint and your firewall/proxy must hand back requests to the HTTP port of the LiveCycle Data Services server, not the HTTPS port.

Testing channel and endpoint performance under load

Use the LiveCycle Data Services load testing tool in a Java client application to load test LiveCycle Data Services NIO- and servlet-based channels/endpoints running in a LiveCycle Data Services web application.

The LiveCycle Data Services installation includes the load testing tool with sample test driver classes and a readme file in the install_root/resources/load-testing-tool directory.

Managing session data

Instances of the FlexClient, MessageClient, and FlexSession classes on the LiveCycle Data Services server represent a Flex application and its connections to the server. You can use these objects to manage synchronization between a Flex application and the server.

FlexClient, MessageClient, and FlexSession objects

The FlexClient object

Every Flex application, written in MXML or ActionScript, is eventually compiled into a SWF file. When the SWF file connects to the LiveCycle Data Services server, a flex.messaging.client.FlexClient object is created to represent that SWF file on the server. SWF files and FlexClient instances have a one-to-one mapping. In this mapping, every FlexClient instance has a unique identifier named `id`, which the LiveCycle Data Services server generates. An ActionScript singleton class, mx.messaging.FlexClient, is also created for the Flex application to access its unique FlexClient `id`.

The MessageClient object

If a Flex application contains a Consumer component (flex.messaging.Consumer), the server creates a corresponding flex.messaging.MessageClient instance that represents the subscription state of the Consumer component. Every MessageClient has a unique identifier named `clientId`. The LiveCycle Data Services server can automatically generate the `clientId` value, but the Flex application can also set the value in the `Consumer.clientId` property before calling the `Consumer.subscribe()` method.

The FlexSession object

A FlexSession object represents the connection between the Flex application and the LiveCycle Data Services server. Its life cycle depends on the underlying protocol, which is determined by the channels and endpoints used on the client and server, respectively.

If you use an RTMP channel in the Flex application, the FlexSession on the LiveCycle Data Services server is scoped to the underlying RTMP connection from the single SWF file. The server is immediately notified when the underlying SWF file is disconnected because RTMP provides a duplex socket connection between the SWF file and the LiveCycle Data Services server. RTMP connections are created for individual SWF files, so when the connection is closed, the associated FlexSession is invalidated.

If an HTTP-based channel, such as AMFChannel or HTTPChannel, is used in the Flex application, the FlexSession on the LiveCycle Data Services server is scoped to the browser and wraps an HTTP session. If the HTTP-based channel connects to a servlet-based endpoint, the underlying HTTP session is a J2EE HttpSession object. If the channel connects to an NIO-based endpoint, the underlying HTTP session supports the FlexSession API, but it is disjointed from the application server HttpSession object.

The relationship between FlexClient, MessageClient, and FlexSession classes

A FlexClient object can have one or more FlexSession instances associated with it depending on the channels that the Flex application uses. For example, if the Flex application uses one HTTPChannel, one FlexSession represents the HTTP session created for that HTTPChannel on the LiveCycle Data Services server. If the Flex application uses an HTTPChannel and an RTMPChannel, two FlexSessions are created; one represents the HTTP session and the other represents the RTMP session.

A FlexSession can also have one or more FlexClients associated with it. For example, when a SWF file that uses an HTTPChannel is opened in two tabs, two FlexClient instances are created in the LCDS server (one for each SWF file), but there is only one FlexSession because two tabs share the same underlying HTTP session.

In terms of hierarchy, FlexClient and FlexSession are peers whereas there is a parent-child relationship between FlexClient/FlexSession and MessageClient. A MessageClient is created for every Consumer component in the Flex application. A Consumer must be contained in a single SWF file and it must subscribe over a single channel. Therefore, each MessageClient is associated with exactly one FlexClient and one FlexSession.

If either the FlexClient or the FlexSession is invalidated on the server, it invalidates the MessageClient. This behavior matches the behavior on the client. If you close the SWF file, the client subscription state is invalidated. If you disconnect the channel or it loses connectivity, the Consumer component is unsubscribed.

Event listeners for FlexClient, MessageClient, and FlexSession

The LiveCycle Data Services server provides the following set of event listener interfaces that allow you to execute custom business logic as FlexClient, FlexSession, and MessageClient instances are created and destroyed and as their state changes:

Event listener	Description
FlexClientListener	FlexClientListener supports listening for life cycle events for FlexClient instances.
FlexClientAttributeListener	FlexClientAttributeListener supports notification when attributes are added, replaced, or removed from FlexClient instances.
FlexClientBindingListener	FlexClientBindingListener supports notification when the implementing class is bound or unbound as an attribute to a FlexClient instance.
FlexSessionListener	FlexSessionListener supports listening for life cycle events for FlexSession instances.
FlexSessionAttributeListener	FlexSessionAttributeListener supports notification when attributes are added, replaced, or removed from FlexSession instances.
FlexSessionBindingListener	FlexSessionBindingListener supports notification when the implementing class is bound or unbound as an attribute to a FlexSession instance.
MessageClientListener	MessageClientListener supports listening for life cycle events for MessageClient instances representing Consumer subscriptions.

For more information about these classes, see the Javadoc API documentation.

Log categories for FlexClient, MessageClient, and FlexSession classes

The following server-side log filter categories can be used to track creation, destruction, and other relevant information for FlexClient, MessageClient, and FlexSession:

- Client.FlexClient
- Client.MessageClient
- Endpoint.FlexSession

Using the FlexContext class with FlexSession and FlexClient attributes

The flex.messaging.FlexContext class is a utility class that exposes the current execution context on the LiveCycle Data Services server. It provides access to FlexSession and FlexClient instances associated with the current message being processed. It also provides global context by accessing MessageBroker, ServletContext, and ServletConfig instances.

The following example shows a Java class that calls `FlexContext.getHttpRequest()` to get an HttpServletRequest object and calls `FlexContext.getFlexSession()` to get a FlexSession object. Exposing this class as a remote object makes it accessible to a Flex client application. Place the compiled class in the WEB_INF/classes directory or your LiveCycle Data Services web application.

```
package myROPackage;

import flex.messaging.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionRO {

    public HttpServletRequest request;
    public FlexSession session;

    public SessionRO() {
        request = FlexContext.getHttpRequest();
        session = FlexContext.getFlexSession();
    }

    public String getSessionId() throws Exception {
        String s = new String();
        s = (String) session.getId();
        return s;
    }

    public String getHeader(String h) throws Exception {
        String s = new String();
        s = (String) request.getHeader(h);
        return h + "=" + s;
    }
}
```

The following example shows a Remoting Service destination definition that exposes the SessionRO class as a remote object. You add this destination definition to your Remoting Service configuration file.

```
...
<destination id="myRODestination">
    <properties>
        <source>myROPackage.SessionRO</source>
    </properties>
</destination>
...
```

The following example shows an ActionScript snippet for calling the remote object from a Flex client application. You place this code inside a method declaration.

```
...
    ro = new RemoteObject();
    ro.destination = "myRODestination";
    ro.getSessionId.addEventListener("result", getSessionIdResultHandler);
    ro.getSessionId();
...
```

When you use an RTMP channel/endpoint, you can call the `getClientInfo()` method of the `FlexSession` instance to get the IP address of the remote RTMP client and determine if the connection is an SSL connection, as the `getClientIp()` method in the following example shows. You must cast the `FlexSession` instance to its `RTMPFlexSession` subclass.

```
package features.remoting;
import features.dataservice.paging.Employee;
import flex.messaging.FlexContext;
import flex.messaging.FlexSession;
import flex.messaging.MessageBroker;
import flex.messaging.endpoints.rtmp.ClientInfo;
import flex.messaging.endpoints.rtmp.RTMPFlexSession;
import flex.messaging.messages.AsyncMessage;
import flex.messaging.util.UUIDUtils;
import javax.servlet.http.HttpServletRequest;
public class EchoService
{
    public String getClientIp()
    {
        FlexSession session = FlexContext.getFlexSession();
        if (session instanceof RTMPFlexSession)
        {
            ClientInfo clientInfo = ((RTMPFlexSession)session).getClientInfo();
            return "Client ip: " + clientInfo.getIp() + " isSecure? " + clientInfo.isSecure();
        }
        return null;
    }
}
```

For more information about FlexContext, see the Javadoc API documentation.

Session life cycle

If you use an RTMP channel in the Flex application, the server is immediately notified when the SWF file is disconnected because RTMP provides a duplex socket connection between the SWF file and the LiveCycle Data Services server. RTMP channels connect to individual SWF files, so when the connection is closed, the associated FlexSession is invalidated. This is not the case with HTTP-based channels because HTTP is a stateless protocol. However, you can use certain techniques in your code to disconnect from HTTP-based channels and invalidate HTTP sessions.

Disconnecting from an HTTP-based channel

Because HTTP is a stateless protocol, when a SWF file is disconnected, notification on the server depends on when the HTTP session times out on the server. When you want a Flex application to notify the LiveCycle Data Services server that it is closing, you call the `disconnectAll()` method on the ChannelSet from JavaScript.

In your Flex application, expose a function for JavaScript to call, as the following example shows:

```
<mx:Application creationComplete="init();">
...
<mx:Script>
<! [CDATA[
    import flash.external.ExternalInterface;
    private function init():void
    {
        if (ExternalInterface.available)
        {
            ExternalInterface.addCallback("disconnectAll", disconnectAll);
        }
    }

    // This function will be called by JavaScript
    private function disconnectAll():void
    {
        // Here you get channelSet of your component and call disconnectAll.
        // For example: producer.channelSet.disconnectAll();
    }
}
...
</mx:Script>
</mx:Application>
```

Next, in the `body` element of the HTML file that wraps your SWF file, you specify an `onunload()` function, as the following example shows:

```
<body ... onunload="disconnectAll()" ...>
```

Finally, you define the JavaScript `disconnectAll()` function that calls the ActionScript `disconnectAll()` method, as the following example shows:

```
<script language='javascript' ...>

    function disconnectAll()
    {
        var answer = confirm("Disconnect All?")
        if (answer)
        {
            // Here you'll need to provide the id of your Flex swf
            document.getElementById("yourFlexSwfId").disconnectAll();

            // And it is important that you have some alert or confirm
            // here to make sure disconnectAll is called before the
            // browser window is closed.
            alert("Disconnected!")
        }
    }
</script>
```

Invalidating an HTTP session

Another complication with using the HTTP protocol is that multiple SWF files share the same HTTP session. When one SWF file disconnects, LiveCycle Data Services cannot invalidate the HTTP session. In this scenario, the default behavior on the server is to leave the current session in place for other applications or pages that are using the HTTP session. However, you can use the optional `invalidate-session-on-disconnect` configuration property in a channel definition in the `services-config.xml` file to invalidate the session, as the following example shows:

```
<channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://servername:port/contextroot/messagebroker/amf"
        class="flex.messaging.endpoints.AMFEEndpoint"/>
    <properties>
        <invalidate-session-on-disconnect>true</invalidate-session-on-disconnect>
    </properties>
</channel>
```

Data serialization

LiveCycle Data Services and Flex provide functionality for serializing data to and from ActionScript objects on the client and Java objects on the server, as well as serializing to and from ActionScript objects on the client and SOAP and XML schema types. Basic types are automatically serialized and deserialized. When working with the Data Management Service or Remoting Service, you can explicitly map custom ActionScript types to custom Java types.

Serialization between ActionScript and Java

LiveCycle Data Services and Flex let you serialize data between ActionScript (AMF 3) and Java in both directions.

Data conversion from ActionScript to Java

When method parameters send data from a Flex application to a Java object, the data is automatically converted from an ActionScript data type to a Java data type. When LiveCycle Data Services searches for a suitable method on the Java object, it uses further, more lenient conversions to find a match.

Simple data types on the client, such as Boolean and String values, typically exactly match a remote API. However, Flex attempts some simple conversions when searching for a suitable method on a Java object.

An ActionScript Array can index entries in two ways. A *strict Array* is one in which all indices are Numbers. An *associative Array* is one in which at least one index is based on a String. It is important to know which type of Array you are sending to the server, because it changes the data type of parameters that are used to invoke a method on a Java object. A *dense Array* is one in which all numeric indices are consecutive, with no gap, starting from 0 (zero). A *sparse Array* is one in which there are gaps between the numeric indices; the Array is treated like an object and the numeric indices become properties that are serialized into a java.util.Map object to avoid sending many null entries.

The following table lists the supported ActionScript (AMF 3) to Java conversions for simple data types:

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
Array (dense)	java.util.List	<p>java.util.Collection, <i>Object</i>[] (native array)</p> <p>If the type is an interface, it is mapped to the following interface implementations</p> <ul style="list-style-type: none"> • List becomes ArrayList • SortedSet becomes TreeSet • Set becomes HashSet • Collection becomes ArrayList <p>A new instance of a custom Collection implementation is bound to that type.</p>
Array (sparse)	java.util.Map	java.util.Map
Boolean String of "true" or "false"	java.lang.Boolean	Boolean, boolean, String
flash.utils.ByteArray	byte []	
flash.utils.IExternalizable	java.io.Externalizable	
Date	java.util.Date (formatted for Coordinated Universal Time (UTC))	java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Time, java.sql.Date
int/uint	java.lang.Integer	java.lang.Double, java.lang.Long, java.lang.Float, java.lang.Integer, java.lang.Short, java.lang.Byte, java.math.BigDecimal, java.math.BigInteger, String, primitive types of double, long, float, int, short, byte
null	null	primitives
Number	java.lang.Double	java.lang.Double, java.lang.Long, java.lang.Float, java.lang.Integer, java.lang.Short, java.lang.Byte, java.math.BigDecimal, java.math.BigInteger, String, 0 (zero) if null is sent, primitive types of double, long, float, int, short, byte
Object (generic)	java.util.Map	If a Map interface is specified, creates a new java.util.HashMap for java.util.Map and a new java.util.TreeMap for java.util.SortedMap.
String	java.lang.String	java.lang.String, java.lang.Boolean, java.lang.Number, java.math.BigInteger, java.math.BigDecimal, char[], enum, any primitive number type
typed Object	typed Object when you use [RemoteClass] metadata tag that specifies remote class name. Bean type must have a public no args constructor.	typed Object

ActionScript type (AMF 3)	Deserialization to Java	Supported Java type binding
undefined	null	null for Object, default values for primitives
XML	org.w3c.dom.Document	org.w3c.dom.Document
XMLDocument (legacy XML type)	org.w3c.dom.Document	org.w3c.dom.Document You can enable legacy XML support for the XMLDocument type on any channel defined in the services-config.xml file. This setting is only important for sending data from the server back to the client; it controls how org.w3c.dom.Document instances are sent to ActionScript. For more information, see " Configuring AMF serialization on a channel " on page 84.

Primitive values cannot be set to null in Java. When passing Boolean and Number values from the client to a Java object, Flex interprets null values as the default values for primitive types; for example, 0 for double, float, long, int, short, byte, \u0000 for char, and false for boolean. Only primitive Java types get default values.

LiveCycle Data Services handles java.lang.Throwable objects like any other typed object. They are processed with rules that look for public fields and bean properties, and typed objects are returned to the client. The rules are like normal bean rules except they look for getters for read-only properties. This lets you get more information from a Java exception. If you require legacy behavior for Throwable objects, you can set the `legacy-throwable` property to true on a channel; for more information, see "[Configuring AMF serialization on a channel](#)" on page 84.

You can pass strict Arrays as parameters to methods that expect an implementation of the java.util.Collection or native Java Array APIs.

A Java Collection can contain any number of Object types, whereas a Java Array requires that entries are the same type (for example, `java.lang.Object[]`, and `int[]`).

LiveCycle Data Services also converts ActionScript strict Arrays to appropriate implementations for common Collection API interfaces. For example, if an ActionScript strict Array is sent to the Java object method `public void addProducts(java.util.Set products)`, LiveCycle Data Services converts it to a `java.util.HashSet` instance before passing it as a parameter, because `HashSet` is a suitable implementation of the `java.util.Set` interface. Similarly, LiveCycle Data Services passes an instance of `java.util.TreeSet` to parameters typed with the `java.util.SortedSet` interface.

LiveCycle Data Services passes an instance of `java.util.ArrayList` to parameters typed with the `java.util.List` interface and any other interface that extends `java.util.Collection`. Then these types are sent back to the client as `mx.collections.ArrayCollection` instances. If you require normal ActionScript Arrays sent back to the client, you must set the `legacy-collection` element to true in the `serialization` section of a channel-definition's properties; for more information, see "[Configuring AMF serialization on a channel](#)" on page 84.

Explicitly mapping ActionScript and Java objects

For Java objects that LiveCycle Data Services does not handle implicitly, LiveCycle Data Services uses value objects, also known as transfer objects, to send data between client and server. For Java objects on the server side, values found in public bean properties with get/set methods and public variables are sent to the client as properties on an Object. Private properties, constants, static properties, and read-only properties are not serialized. For ActionScript objects on the client side, public properties defined with the get/set accessors and public variables are sent to the server.

LiveCycle Data Services uses the standard Java class, `java.beans.Introspector`, to get property descriptors for a Java bean class. It also uses reflection to gather public fields on a class. It uses bean properties in preference to fields. The Java and ActionScript property names should match. Native Flash Player code determines how ActionScript classes are introspected on the client.

In the ActionScript class, you use the `[RemoteClass(alias=" ")]` metadata tag to create an ActionScript object that maps directly to the Java object. The ActionScript class to which data is converted must be used or referenced in the MXML file for it to be linked into the SWF file and available at run time. A good way to do this is by casting the result object, as the following example shows:

```
var result:MyClass = MyClass(event.result);
```

The class itself should use strongly typed references so that its dependencies are also linked.

The following example shows the source code for an ActionScript class that uses the `[RemoteClass(alias=" ")]` metadata tag:

```
package samples.contact {  
    [Bindable]  
    [RemoteClass(alias="samples.contact.Contact")]  
    public class Contact {  
        public var contactId:int;  
  
        public var firstName:String;  
  
        public var lastName:String;  
  
        public var address:String;  
  
        public var city:String;  
  
        public var state:String;  
  
        public var zip:String;  
    }  
}
```

You can use the `[RemoteClass]` metadata tag without an alias if you do not map to a Java object on the server, but you do send back your object type from the server. Your ActionScript object is serialized to a special Map object when it is sent to the server, but the object returned from the server to the clients is your original ActionScript type.

To restrict a specific property from being sent to the server from an ActionScript class, use the `[Transient]` metadata tag above the declaration of that property in the ActionScript class.

Data conversion from Java to ActionScript

An object returned from a Java method is converted from Java to ActionScript. LiveCycle Data Services also handles objects found within objects. LiveCycle Data Services implicitly handles the Java data types in the following table.

Java type	ActionScript type (AMF 3)
enum (JDK 1.5)	String
java.lang.String	String
java.lang.Boolean, boolean	Boolean
java.lang.Integer, int	int If value < 0xF0000000 value > 0xFFFFFFFF, the value is promoted to Number due to AMF encoding requirements.
java.lang.Short, short	int If i < 0xF0000000 i > 0xFFFFFFFF, the value is promoted to Number.

Java type	ActionScript type (AMF 3)
java.lang.Byte, byte[]	int If $i < 0x00000000 \parallel i > 0xFFFFFFFF$, the value is promoted to Number.
java.lang.Byte[]	flash.utils.ByteArray
java.lang.Double, double	Number
java.lang.Long, long	Number
java.lang.Float, float	Number
java.lang.Character, char	String
java.lang.Character[], char[]	String
java.math.BigInteger	String
java.math.BigDecimal	String
java.util.Calendar	Date Dates are sent in the Coordinated Universal Time (UTC) time zone. Clients and servers must adjust time accordingly for time zones.
java.util.Date	Date Dates are sent in the UTC time zone. Clients and servers must adjust time accordingly for time zones.
java.util.Collection (for example, java.util.ArrayList)	mx.collections.ArrayCollection
java.lang.Object[]	Array
java.util.Map	Object (untyped). For example, a java.util.Map[] is converted to an Array (of Objects).
java.util.Dictionary	Object (untyped)
org.w3c.dom.Document	XML object
null	null
java.lang.Object (other than previously listed types)	Typed Object Objects are serialized using Java bean introspection rules and also include public fields. Fields that are static, transient, or nonpublic, as well as bean properties that are nonpublic or static, are excluded.

Note: You can enable legacy XML support for the flash.xml.XMLDocument type on any channel that is defined in the services-config.xml file. In Flex 1.5, java.util.Map was sent as an associative or ECMA Array. This is no longer a recommended practice. You can enable legacy Map support to associative Arrays, but Adobe recommends against doing this.

The following table contains type mappings between types that are specific to Apache Axis and ActionScript types for RPC-encoded web services:

Apache Axis type	ActionScript type
Map	Object

Apache Axis type	ActionScript type
RowSet	Can only receive RowSets; can't send them.
Document	Can only receive Documents; can't send them.
Element	flash.xml.XMLNode

Configuring AMF serialization on a channel

You can support legacy AMF type serialization used in earlier versions of Flex and configure other serialization properties in channel definitions in the services-config.xml file.

The following table describes the properties that you can set in the `<serialization>` element of a channel definition:

Property	Description
enable-small-messages	Default value is <code>true</code> . If enabled, messages are sent using an alternative smaller form if one is available and the endpoint supports it. When you set the value of this property to <code>false</code> , messages include headers including <code>messageId</code> , <code>timestamp</code> , <code>correlationId</code> , and <code>destination</code> , which can be useful for debugging. When you disable small messages, enable debug logging to include messages in server logs. For information, see “ Server-side logging ” on page 25.
ignore-property-errors	Default value is <code>true</code> . Determines if the endpoint should throw an error when an incoming client object has unexpected properties that cannot be set on the server object.
include-read-only	Default value is <code>false</code> . Determines if read-only properties should be serialized back to the client.
log-property-errors	Default value is <code>false</code> . When <code>true</code> , unexpected property errors are logged.
legacy-collection	Default value is <code>false</code> . When <code>true</code> , instances of <code>java.util.Collection</code> are returned to the client as ActionScript Array objects instead of <code>mx.collections.ArrayCollection</code> objects. When <code>true</code> , during client to server deserialization, instances of ActionScript Array objects are deserialized into Java List objects instead of Java Object arrays.
legacy-map	Default value is <code>false</code> . When <code>true</code> , <code>java.util.Map</code> instances are serialized as an ECMA Array or associative array instead of an anonymous Object.
legacy-xml	Default value is <code>false</code> . When <code>true</code> , <code>org.w3c.dom.Document</code> instances are serialized as <code>flash.xml.XMLDocument</code> instances instead of intrinsic XML (E4X capable) instances.
legacy-throwable	Default value is <code>false</code> . When <code>true</code> , <code>java.lang.Throwable</code> instances are serialized as AMF status-info objects (instead of normal bean serialization, including read-only properties).
legacy-externalizable	Default value is <code>false</code> . When <code>true</code> , <code>java.io.Externalizable</code> types (that extend standard Java classes like <code>Date</code> , <code>Number</code> , <code>String</code>) are not serialized as custom objects (for example, <code>MyDate</code> is serialized as <code>Date</code> instead of <code>MyDate</code>). Note that this setting overwrites any other legacy settings. For example, if <code>legacy-collection</code> is <code>true</code> but the collection implements <code>java.io.Externalizable</code> , the collection is returned as custom object without taking the <code>legacy-collection</code> value into account.

Property	Description
type-marshaller	<p>Specifies an implementation of <code>flex.messaging.io.TypeMarshaller</code> that translates an object into an instance of a desired class. Used when invoking a Java method or populating a Java instance and the type of the input object from deserialization (for example, an ActionScript anonymous Object is always serialized as a <code>java.util.HashMap</code>) doesn't match the destination API (for example, <code>java.util.SortedMap</code>). Thus, the type can be marshalled into the desired type.</p> <p>The <code>flex.messaging.io.amf.translator.ASTranslator</code> class is an implementation of <code>TypeMarshaller</code> that you can use as an example for your own <code>TypeMarshaller</code> implementations.</p>
restore-references	<p>Default value is <code>false</code>. An advanced switch to make the deserializer keep track of object references when a type translation has to be made; for example, when an anonymous Object is sent for a property of type <code>java.util.SortedMap</code>, the Object is first serialized to a <code>java.util.Map</code> as normal, and then translated to a suitable implementation of <code>SortedMap</code> (such as <code>java.util.TreeMap</code>). If other objects pointed to the same anonymous Object in an object graph, this setting restores those references instead of creating <code>SortedMap</code> implementations everywhere. Notice that setting this property to <code>true</code> can slow down performance significantly for large amounts of data.</p>
instantiate-types	<p>Default value is <code>true</code>. Advanced switch that when set to <code>false</code> stops the deserializer from creating instances of strongly typed objects and instead retains the type information and deserializes the raw properties in a Map implementation, specifically <code>flex.messaging.io.ASObject</code>. Notice that any classes under <code>flex.*</code> package are always instantiated.</p>

Using custom serialization between ActionScript and Java

If the standard mechanisms for serializing and deserializing data between ActionScript on the client and Java on the server do not meet your needs, you can write your own serialization scheme. You implement the ActionScript-based `flash.utils.IExternalizable` interface on the client and the corresponding Java-based `java.io.Externalizable` interface on the server.

A typical reason to use custom serialization is to avoid passing all of the properties of either the client-side or server-side representation of an object across the network tier. When you implement custom serialization, you can code your classes so that specific properties that are client-only or server-only are not passed over the wire. When you use the standard serialization scheme, all public properties are passed back and forth between the client and the server.

On the client side, the identity of a class that implements the `flash.utils.IExternalizable` interface is written in the serialization stream. The class serializes and reconstructs the state of its instances. The class implements the `writeExternal()` and `readExternal()` methods of the `IExternalizable` interface to get control over the contents and format of the serialization stream, but not the class name or type, for an object and its supertypes. These methods supersede the native AMF serialization behavior. These methods must be symmetrical with their remote counterpart to save the state of the class.

On the server side, a Java class that implements the `java.io.Externalizable` interface performs functionality that is analogous to an ActionScript class that implements the `flash.utils.IExternalizable` interface.

Note: You should not use types that implement the `IExternalizable` interface with the `HTTPChannel` if precise by-reference serialization is required. When you do this, references between recurring objects are lost and appear to be cloned at the endpoint.

Client-side Product class

The following example shows the complete source code for the client (ActionScript) version of a Product class that maps to a Java-based Product class on the server. The client Product class implements the IExternalizable interface and the server Product class implements the Externalizable interface.

```
// Product.as
package samples.externalizable {

    import flash.utils.IExternalizable;
    import flash.utils.IDataInput;
    import flash.utils.IDataOutput;

    [RemoteClass(alias="samples.externalizable.Product")]
    public class Product implements IExternalizable {
        public function Product(name:String=null) {
            this.name = name;
        }

        public var id:int;
        public var name:String;
        public var properties:Object;
        public var price:Number;

        public function readExternal(input:IDataInput):void {
            name = input.readObject() as String;
            properties = input.readObject();
            price = input.readFloat();
        }

        public function writeExternal(output:IDataOutput):void {
            output.writeObject(name);
            output.writeObject(properties);
            output.writeFloat(price);
        }
    }
}
```

The client Product class uses two kinds of serialization. It uses the standard serialization that is compatible with the java.io.Externalizable interface and AMF 3 serialization. The following example shows the `writeExternal()` method of the client Product class, which uses both types of serialization:

```
public function writeExternal(output:IDataOutput):void {
    output.writeObject(name);
    output.writeObject(properties);
    output.writeFloat(price);
}
```

As the following example shows, the `writeExternal()` method of the server Product class is almost identical to the client version of this method:

```
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(name);
    out.writeObject(properties);
    out.writeFloat(price);
}
```

In the `writeExternal()` method of the client Product class, the `flash.utils.IDataOutput.writeFloat()` method is an example of standard serialization methods that meet the specifications for the Java `java.io.DataInput.readFloat()` methods for working with primitive types. This method sends the `price` property, which is a `Float`, to the server Product.

The examples of AMF 3 serialization in the `writeExternal()` method of the client Product class is the call to the `flash.utils.IDataOutput.writeObject()` method, which maps to the `java.io.ObjectInput.readObject()` method call in the server Product class `readExternal()` method. The `flash.utils.IDataOutput.writeObject()` method sends the `properties` property, which is an `Object`, and the `name` property, which is a `String`, to the server Product. This is possible because the `AMFChannel` endpoint has an implementation of the `java.io.ObjectInput` interface that expects data sent from the `writeObject()` method to be formatted as AMF 3.

In turn, when the `readObject()` method is called in the server Product's `readExternal()` method, it uses AMF 3 deserialization; this is why the ActionScript version of the `properties` value is assumed to be of type `Map` and `name` is assumed to be of type `String`.

Server-side Product class

The following example shows the complete source code of the server Product class:

```
// Product.java
package samples.externalizable;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Map;

/**
 * This Externalizable class requires that clients sending and
 * receiving instances of this type adhere to the data format
 * required for serialization.
 */
public class Product implements Externalizable {
    private String inventoryId;
    public String name;
    public Map properties;
    public float price;

    public Product()
    {

    }

    /**
     * Local identity used to track third party inventory. This property is
     * not sent to the client because it is server-specific.
     * The identity must start with an 'X'.
     */
    public String getInventoryId() {
        return inventoryId;
    }

    public void setInventoryId(String inventoryId) {
        if (inventoryId != null && inventoryId.startsWith("X"))
    }
}
```

```

        this.inventoryId = inventoryId;
    }
} else
{
    throw new IllegalArgumentException("3rd party product
        inventory identities must start with 'X'");
}
}

/**
 * Deserializes the client state of an instance of ThirdPartyProxy
 * by reading in String for the name, a Map of properties
 * for the description, and
 * a floating point integer (single precision) for the price.
 */
public void readExternal(ObjectInput in) throws IOException,
    ClassNotFoundException {
    // Read in the server properties from the client representation.
    name = (String)in.readObject();
    properties = (Map)in.readObject();
    price = in.readFloat();
    setInventoryId(lookupInventoryId(name, price));
}
/**
 * Serializes the server state of an instance of ThirdPartyProxy
 * by sending a String for the name, a Map of properties
 * String for the description, and a floating point
 * integer (single precision) for the price. Notice that the inventory
 * identifier is not sent to external clients.
 */
public void writeExternal(ObjectOutput out) throws IOException {
    // Write out the client properties from the server representation
    out.writeObject(name);
    out.writeObject(properties);
    out.writeFloat(price);
}

private static String lookupInventoryId(String name, float price) {
    String inventoryId = "X" + name + Math.rint(price);
    return inventoryId;
}
}

```

The following example shows the `readExternal()` method of the server Product class:

```

public void readExternal(ObjectInput in) throws IOException,
    ClassNotFoundException {
    // Read in the server properties from the client representation.
    name = (String)in.readObject();
    properties = (Map)in.readObject();
    price = in.readFloat();
    setInventoryId(lookupInventoryId(name, price));
}

```

The `writeExternal()` method of the client Product class does not send the `id` property to the server during serialization because it is not useful to the server version of the Product object. Similarly, the `writeExternal()` method of the server Product class does not send the `inventoryId` property to the client because it is a server-specific property.

Notice that the names of a Product properties are not sent during serialization in either direction. Because the state of the class is fixed and manageable, the properties are sent in a well-defined order without their names, and the `readExternal()` method reads them in the appropriate order.

Remote object class

The following example shows the source code of the Java class, `ProductRegistry`, which is called with the `RemoteObject` component on the client:

```
package example.externalizable;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
/**
 * A simple registry to manage instances of Product for an example of the
 * Externalizable API for custom serialization.
 */
public class ProductRegistry
{
    public ProductRegistry()
    {
        registry = Collections.synchronizedMap(new HashMap());
        Product p = new Product();
        p.name = "Example Widget";
        p.description = "The right widget for any problem.";
        p.price = 350;
        registerProduct(p);
        p = new Product();
        p.name = "Example Gift";
        p.description = "The perfect gift for any occasion.";
        p.price = 225;
        registerProduct(p);
    }
    public void registerProduct(Product product)
    {
        registry.put(product.getId(), product);
    }
    public Collection getProducts()
    {
        return registry.values();
    }
    private Map registry;
}
```

Destination configuration

The following XML snippet shows the `ProductRegistry` destination in the `remoting-config.xml` file:

```
<destination id="ProductRegistry">
    <properties>
        <source>example.externalizable.ProductRegistry</source>
        <scope>application</scope>
    </properties>
</destination>
```

MXML application code

The following example shows the MXML application that calls the ProductRegistry destination on the server:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.controls.Alert;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import example.externalizable.Product;
            [Bindable]
            public var products:ArrayCollection

            // Create a compile time dependency on Product class.
            private static var dep:Product;
            public function resultHandler(event:ResultEvent):void
            {
                products = event.result as ArrayCollection;
            }
            public function faultHandler(event:FaultEvent):void
            {
                Alert.show("Fault", event.fault.toString());
            }
        ]]>
    </mx:Script>
    <mx:RemoteObject id="remoteObject" destination="ProductRegistry"
        fault="faultHandler(event)"
        result="resultHandler(event)" />
    <mx:Panel title="Externalizable Example" height="400" width="600"
        paddingTop="10" paddingLeft="10" paddingRight="10">
```

```
<mx:DataGrid id="grid" width="100%" rowCount="5" dataProvider="{products}">
    <mx:columns>
        <mx:DataGridColumn dataField="name" headerText="Name"/>
        <mx:DataGridColumn dataField="price" headerText="Price"/>
    </mx:columns>
</mx:DataGrid>
<mx:Form width="100%">
    <mx:FormItem label="Name">
        <mx:Label text="{grid.selectedItem.name}" />
    </mx:FormItem>
    <mx:FormItem label="Price">
        <mx:Label text="{grid.selectedItem.price}" />
    </mx:FormItem>
    <mx:FormItem label="Description">
        <mx:TextArea text="{grid.selectedItem.description}" />
    </mx:FormItem>
</mx:Form>
<mx:Button label="Get Products" click="remoteObject.getProducts() " />
</mx:Panel>
</mx:Application>
```

Serialization between ActionScript and web services

Default encoding of ActionScript data

The following table shows the default encoding mappings from ActionScript 3 types to XML schema complex types.

XML schema definition	Supported ActionScript 3 types	Notes
Top-level elements		
xsd:element nillable == true	Object	If input value is null, encoded output is set with the xsi:nil attribute.
xsd:element fixed != null	Object	Input value is ignored and fixed value is used instead.
xsd:element default != null	Object	If input value is null, this default value is used instead.
Local elements		
xsd:element maxOccurs == 0	Object	Input value is ignored and omitted from encoded output.

xsd:element maxOccurs == 1	Object	Input value is processed as a single entity. If the associated type is a SOAP-encoded array, then arrays and mx.collection.IList implementations pass through intact and are handled as a special case by the SOAP encoder for that type.
xsd:element maxOccurs > 1	Object	Input value should be iterable (such as an array or mx.collections.IList implementation), although noniterable values are wrapped before processing. Individual items are encoded as separate entities according to the definition.
xsd:element minOccurs == 0	Object	If input value is undefined or null, encoded output is omitted.

The following table shows the default encoding mappings from ActionScript 3 types to XML schema built-in types.

XML schema type	Supported ActionScript 3 types	Notes
xsd:anyType xsd:anySimpleType	Object	Boolean -> xsd:boolean ByteArray -> xsd:base64Binary Date -> xsd:dateTime int -> xsd:int Number -> xsd:double String -> xsd:string uint -> xsd:unsignedInt
xsd:base64Binary	flash.utils.ByteArray	mx.utils.Base64Encoder is used (without line wrapping).
xsd:boolean	Boolean Number Object	Always encoded as true or false. Number == 1 then true, otherwise false. Object.toString() == "true" or "1" then true, otherwise false.
xsd:byte xsd:unsignedByte	Number String	String first converted to Number.
xsd:date	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
xsd:dateTime	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
xsd:decimal	Number String	Number.toString() is used. Note that Infinity, -Infinity, and NaN are invalid for this type. String first converted to Number.

XML schema type	Supported ActionScript 3 types	Notes
xsd:double	Number	Limited to range of Number.
	String	String first converted to Number.
xsd:duration	Object	Object. <code>toString()</code> is called.
xsd:float	Number	Limited to range of Number.
	String	String first converted to Number.
xsd:gDay	Date	<code>Date.getUTCDate()</code> is used.
	Number	Number used directly for day.
	String	String parsed as Number for day.
xsd:gMonth	Date	<code>Date.getUTCMonth()</code> is used.
	Number	Number used directly for month.
	String	String parsed as Number for month.
xsd:gMonthDay	Date	<code>Date.getUTCMonth()</code> and <code>Date.getUTCDate()</code> are used.
	String	String parsed for month and day portions.
xsd:gYear	Date	<code>Date.getUTCFullYear()</code> is used.
	Number	Number used directly for year.
	String	String parsed as Number for year.
xsd:gYearMonth	Date	<code>Date.getUTCFullYear()</code> and <code>Date.getUTCMonth()</code> are used.
	String	String parsed for year and month portions.
xsd:hexBinary	flash.utils.ByteArray	mx.utils.HexEncoder is used.
xsd:integer and derivatives: xsd:negativeInteger xsd:nonNegativeInteger xsd:positiveInteger xsd:nonPositiveInteger	Number	Limited to range of Number.
	String	String first converted to Number.
xsd:int	Number	String first converted to Number.
xsd:unsignedInt	String	
xsd:long	Number	String first converted to Number.
xsd:unsignedLong	String	
xsd:short	Number	String first converted to Number.
xsd:unsignedShort	String	

XML schema type	Supported ActionScript 3 types	Notes
xsd:string and derivatives: xsd:ID xsd:IDREF xsd:IDREFS xsd:ENTITY xsd:ENTITIES xsd:language xsd:Name xsd:NCName xsd:NMTOKEN xsd:NMTOKENS xsd:normalizedString xsd:token	Object	Object. <code>toString()</code> is invoked.
xsd:time	Date Number String	Date UTC accessor methods are used. Number used to set Date.time. String assumed to be preformatted and encoded as is.
xsi:nil	null	If the corresponding XML schema element definition has <code>minOccurs > 0</code> , a <code>null</code> value is encoded by using <code>xsi:nil</code> ; otherwise the element is omitted entirely.

The following table shows the default mapping from ActionScript 3 types to SOAP-encoded types.

SOAPENC type	Supported ActionScript 3 types	Notes
soapenc:Array	Array <code>mx.collections.IList</code>	SOAP-encoded arrays are special cases and are supported only with RPC-encoded web services.
soapenc:base64	<code>flash.utils.ByteArray</code>	Encoded in the same manner as <code>xsd:base64Binary</code> .
soapenc:*	Object	Any other SOAP-encoded type is processed as if it were in the XSD namespace based on the <code>localName</code> of the type's QName.

Default decoding of XML schema and SOAP to ActionScript 3

The following table shows the default decoding mappings from XML schema built-in types to ActionScript 3 types.

XML schema type	Decoded ActionScript 3 types	Notes
xsd:anyType xsd:anySimpleType	String Boolean Number	If content is empty -> xsd:string. If content cast to Number and value is NaN; or if content starts with "0" or "-0", or if content ends with "E": then, if content is "true" or "false" -> xsd:boolean otherwise -> xsd:string. Otherwise content is a valid Number and thus -> xsd:double.
xsd:base64Binary	flash.utils.ByteArray	mx.utils.Base64Decoder is used.
xsd:boolean	Boolean	If content is "true" or "1" then true, otherwise false.
xsd:date	Date	If no time zone information is present, local time is assumed.
xsd:dateTime	Date	If no time zone information is present, local time is assumed.
xsd:decimal	Number	Content is created via Number(content) and is thus limited to the range of Number. Number.NaN, Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY are not allowed.
xsd:double	Number	Content is created via Number(content) and is thus limited to the range of Number.
xsd:duration	String	Content is returned with whitespace collapsed.
xsd:float	Number	Content is converted through Number(content) and is thus limited to the range of Number.
xsd:gDay	uint	Content is converted through uint(content).
xsd:gMonth	uint	Content is converted through uint(content).
xsd:gMonthDay	String	Content is returned with whitespace collapsed.
xsd:gYear	uint	Content is converted through uint(content).
xsd:gYearMonth	String	Content is returned with whitespace collapsed.
xsd:hexBinary	flash.utils.ByteArray	mx.utils.HexDecoder is used.

XML schema type	Decoded ActionScript 3 types	Notes
xsd:integer and derivatives: xsd:byte xsd:int xsd:long xsd:negativeInteger xsd:nonNegativeInteger xsd:nonPositiveInteger xsd:positiveInteger xsd:short xsd:unsignedByte xsd:unsignedInt xsd:unsignedLong xsd:unsignedShort	Number	Content is decoded via <code>parseInt()</code> . Number.NaN, Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY are not allowed.
xsd:string and derivatives: xsd:ID xsd:IDREF xsd:IDREFS xsd:ENTITY xsd:ENTITIES xsd:language xsd:Name xsd:NCName xsd:NMTOKEN xsd:NMTOKENS xsd:normalizedString xsd:token	String	The raw content is simply returned as a string.
xsd:time	Date	If no time zone information is present, local time is assumed.
xsi:nil	null	

The following table shows the default decoding mappings from SOAP-encoded types to ActionScript 3 types.

SOAPENC type	Decoded ActionScript type	Notes
soapenc:Array	Array <code>mx.collections.ArrayCollection</code>	SOAP-encoded arrays are special cases. If <code>makeObjectsBindable</code> is true, the result is wrapped in an <code> ArrayCollection</code> ; otherwise a simple array is returned.
soapenc:base64	<code>flash.utils.ByteArray</code>	Decoded in the same manner as <code>xsd:base64Binary</code> .
soapenc:*	Object	Any other SOAP-encoded type is processed as if it were in the XSD namespace based on the <code>localName</code> of the type's QName.

The following table shows the default decoding mappings from custom data types to ActionScript 3 data types.

Custom type	Decoded ActionScript 3 type	Notes
Apache Map <code>http://xml.apache.org/xml-soap:Map</code>	Object	SOAP representation of <code>java.util.Map</code> . Keys must be representable as strings.
Apache Rowset <code>http://xml.apache.org/xml-soap:Rowset</code>	Array of objects	
ColdFusion QueryBean <code>http://rpc.xml.coldfusion:QueryBean</code>	Array of objects <code>mx.collections.ArrayCollection of objects</code>	If <code>makeObjectsBindable</code> is true, the resulting array is wrapped in an <code> ArrayCollection</code> .

XML Schema element support

The following XML schema structures or structure attributes are only partially implemented in Flex 3:

```
<choice>
<all>
<union>
```

The following XML Schema structures or structure attributes are ignored and are not supported in Flex 3:

```
<attribute use="required"/>

<element
    substitutionGroup="..."
    unique="..."
    key="..."
    keyref="..."
    field="..."
    selector="..."/>

<simpleType>
    <restriction>
        <minExclusive>
        <minInclusive>
        <maxExclusive>
        <maxInclusive>
        <totalDigits>
        <fractionDigits>
        <length>
        <minLength>
        <maxLength>
        <enumeration>
        <whiteSpace>
        <pattern>
    </restriction>
</simpleType>

<complexType
    final="..."
    block="..."
    mixed="..."
    abstract="..."/>

<any
processContents="..."/>
<annotation>
```

Customizing web service type mapping

When consuming data from a web service invocation, Flex usually creates untyped anonymous ActionScript objects that mimic the XML structure in the body of the SOAP message. If you want Flex to create an instance of a specific class, you can use an mx.rpc.xml.SchemaTypeRegistry object and register a QName object with a corresponding ActionScript class.

For example, suppose you have the following class definition in a file named User.as:

```
package
{
    public class User
    {
        public function User() {}

        public var firstName:String;
        public var lastName:String;
    }
}
```

Next, you want to invoke a `getUser` operation on a web service that returns the following XML:

```
<tns:getUserResponse xmlns:tns="http://example.uri">
  <tns:firstName>Ivan</tns:firstName>
  <tns:lastName>Petrov</tns:lastName>
</tns:getUserResponse>
```

To make sure you get an instance of your `User` class instead of a generic `Object` when you invoke the `getUser` operation, you need the following ActionScript code inside a method in your Flex application:

```
SchemaTypeRegistry.getInstance().registerClass(new QName("http://example.uri",
  "getUserResponse"), User);
```

`SchemaTypeRegistry.getInstance()` is a static method that returns the default instance of the type registry. In most cases, that is all you need. However, this registers a given `QName` with the same ActionScript class across all web service operations in your application. If you want to register different classes for different operations, you need the following code in a method in your application:

```
var qn:QName = new QName("http://the.same", "qname");
var typeReg1:SchemaTypeRegistry = new SchemaTypeRegistry();
var typeReg2:SchemaTypeRegistry = new SchemaTypeRegistry();
typeReg1.registerClass(qn, someClass);
myWS.someOperation.decoder.typeRegistry = typeReg1;

typeReg2.registerClass(qn, anotherClass);
myWS.anotherOperation.decoder.typeRegistry = typeReg2;
```

Using custom web service serialization

There are two approaches to take full control over how ActionScript objects are serialized into XML and how XML response messages are deserialized. The recommended approach is to work directly with E4X.

If you pass an instance of XML as the only parameter to a web service operation, it is passed on untouched as the child of the `<SOAP:Body>` node in the serialized request. Use this strategy when you need full control over the SOAP message. Similarly, when deserializing a web service response, you can set the operation's `resultFormat` property to `e4x`. This returns an `XMLList` object with the children of the `<SOAP:Body>` node in the response message. From there, you can implement the necessary custom logic to create the appropriate ActionScript objects.

The second and more tedious approach is to provide your own implementations of `mx.rpc.soap.ISOAPDecoder` and `mx.rpc.soap.ISOAPEncoder`. For example, if you have written a class called `MyDecoder` that implements `ISOAPDecoder`, you can have the following in a method in your Flex application:

```
myWS.someOperation.decoder = new MyDecoder();
```

When invoking `someOperation`, Flex calls the `decodeResponse()` method of the `MyDecoder` class. From that point on it is up to the custom implementation to handle the full SOAP message and produce the expected ActionScript objects.

Chapter 3: Controlling data traffic

Data throttling

Both BlazeDS and LiveCycle Data Services provide data throttling features that give you control over aspects of the server's underlying messaging system.

You use data throttling to limit the number of server-to-client and client-to-server messages that a Message Service or Data Management Service destination can process per second. You can also set absolute limits for the number of client-to-server and server-to-client messages allowed per second. Without data throttling, the server pushes messages to clients without any indication on how fast the clients are processing messages. Similarly, a high number of messages from Flex clients can overwhelm a server. Throttling is useful when the server could overwhelm slow clients with a high number of messages that they cannot process or the server could be overwhelmed with messages from clients.

For server-to-client messages in LiveCycle Data Services, you can use adaptive throttling to message frequency limits based on the message processing rates of clients. You can also set policies that determine what happens to messages that exceed the specified maximum message limits.

Note: In addition to these standard throttling features, a set of advanced capabilities are available in LiveCycle Data Service only. For more information, see “[Advanced data tuning](#)” on page 106.

Message frequency limits

Control the rate at which a Message Service or Data Management Service destination on the server sends messages to or accepts messages from Flex clients in the following ways:

Message frequency limit	Description
Destination-level frequency	In a destination definition on the server, you can specify server-to-client and client-to-server message frequency limits in the throttle-outbound and throttle-inbound elements, respectively. There are separate settings for controlling message frequency globally and on a per-client basis.
Client-level	On the Flex client side, you can set the <code>maxFrequency</code> property of a Consumer, MultiTopicConsumer, or DataService component to limit the maximum number of messages that the component prefers to receive from the server. If possible, the server accommodates this setting. For MultiTopic Consumer and DataService components, which allow multiple subscriptions, you can set the <code>maxFrequency</code> property on a per subscription basis.
Adaptive	Advanced feature not available in BlazeDS. See “ Advanced data throttling ” on page 109.

Destination-level message frequency

You specify client-to-server and server-to-client message frequency limits for a destination in the destination's `throttle-inbound` and `throttle-outbound` elements. Use the `max-frequency` attribute to set the maximum number of client-to-server or server-to-client messages per second that the destination can process. Use the `max-client-frequency` attribute to set the maximum number of client-to-server or server-to-client messages per second that the destination can process to or from individual clients.

The following example shows `throttle-outbound` and `throttle-inbound` elements with `max-frequency` and `max-client-frequency` attributes:

```
...
<destination id="MyTopic">
    <properties>
        <network>
            <throttle-outbound policy="ERROR"
                max-frequency="100" max-client-frequency="10"/>
            <throttle-inbound policy="IGNORE"
                max-frequency="100" max-client-frequency="10"/>
    ...

```

The `max-frequency` attribute of the `throttle-outbound` element indicates that the destination sends a maximum of 100 messages per second to the entire pool of Flex clients. The `max-client-frequency` attribute indicates that the destination sends a maximum of ten messages per second to individual Flex clients.

The `max-frequency` element of the `throttle-inbound` element indicates that the destination processes a maximum of 100 client-to-server messages per second from the entire pool of Flex clients. The `max-client-frequency` attribute indicates that the destination processes a maximum of ten messages per second from individual Flex clients.

Client-level message frequency

The `maxFrequency` property of a client-side Consumer, MultiTopicConsumer, or DataService component lets you limit the number of messages the component receives from a destination. The server reads this information and ensures that it limits the messages it sends to the maximum number of messages per second specified. The server sends up to the limit, but can send fewer messages depending on how many messages are available on the server to send. The `maxFrequency` property is set when the Consumer, MultiTopicConsumer, or DataService component subscribes.

The following example sets the `maxFrequency` property of a Consumer component that you create in ActionScript:

```
...
<mx:Script>
    <![CDATA[
        var consumer:Consumer = new Consumer();
        consumer.destination = "chat";
        consumer.maxFrequency = 40;
        consumer.subscribe();
    ]]>
</mx:Script>
...
```

You can also set the `maxFrequency` property of a Consumer, MultiTopicConsumer, or DataService component that you create in MXML as the following example shows:

```
...
<mx:Consumer id="consumer" destination="chat" maxFrequency="40"/>
...
```

MultiTopicConsumer message frequency

Unlike the standard Consumer component, the MultiTopicConsumer component supports multiple subscriptions. You can set a default value for the `maxFrequency` property of a MultiTopicConsumer component, and also set separate values for individual subscriptions. The following example shows a MultiTopicConsumer with a default `maxFrequency` value of 40 messages per second. One of its two subscriptions uses the default value, while the other uses a different `maxFrequency` value specified in the third parameter of the `addSubscription()` method. The first two parameters of the `addSubscription()` method specify the subtopic and selector, respectively. The selector value is null in this case.

```
...
<mx:Script>
<! [CDATA[
    var consumer:MultiTopicConsumer = new MultiTopicConsumer();
    ...
    consumer.maxFrequency = 40;
    // For this subscription, use the default maxFrequency of 40.
    consumer.addSubscription("chat.subtopic1");
    // Only for this subscription, overwrite the maxFrequency to 20.
    consumer.addSubscription("chat.subtopic2", null, 20);
    consumer.subscribe();
]]>
</mx:Script>
...
```

For more information about the MultiTopicConsumer component, see “[Multitopic producers and consumers](#)” on page 205.

DataService message frequency

For the DataService component, you specify the default value of the `maxFrequency` property as the following example shows:

```
...
<mx:Script>
<! [CDATA[
    ...
    var ds:DataService = new DataService("Meeting");
    ds.maxFrequency = 20;
    ...
]]>
</mx:Script>
...
```

When you use a DataService component in manual routing mode (manual synchronization of messages), you can specify multiple subscriptions and subscription-level values for the `maxFrequency` property as the following example shows. As with the `MultiTopicConsumer.addSubscription()` method, the `manualSync.consumerAddSubscription()` method takes subtopic, selector, and `maxFrequency` in its three parameters.

```
...
<mx:Script>
<! [CDATA[
    ...
    var ds:DataService = new DataService("Meeting");
    ds.autoSyncEnabled = false;
    ds.manualSync.producerSubtopics.addItem("flex-room");
    // Set the subscription level frequency to 10.
    ds.manualSync.consumerAddSubscription("flex-room", null, 10);
    ds.manualSync.consumerSubscribe();
    ds.fill(...);
    ...
]]>
</mx:Script>
...
```

For more information about manual routing, see “[Manually routing data messages](#)” on page 234.

Adaptive client message frequency

When you enable adaptive client message frequency, the server adjusts the frequency of messages sent from server to client while taking the client's actual message processing rate into account. You specify the maximum number of messages per second that the frequency is incrementally increased or decreased.

Add something about fact that the calculations the server does are different depending the type of endpoint you are using. For example, RTMP versus long polling AMF.

To use adaptive client message frequency, you configure the `adaptive-frequency` property of the `flex-client-outbound-queue-processor` element in the `flex-client` section of the `services-config.xml`, as follows:

- 1 Set the `flex-client-queue-processorclass` value to `"flex.messaging.client.AdvancedOutboundQueueProcessor"`.
- 2 Set the value of the `adaptive-frequency` property to `true`.
- 3 Set the value of the `frequency-step-size` property to the number of messages per second that you want to incrementally increase the message frequency, depending on the server's calculation. This setting is useful when you want to gradually increase the number of messages sent per second.

The following example shows an advanced outbound queue processor configured for adaptive message frequency:

```
...
<flex-client>
    <flex-client-outbound-queue-processor
        class="flex.messaging.client.AdvancedOutboundQueueProcessor">
        <properties>
            <adaptive-frequency>true</adaptive-frequency>
            <frequency-step-size>10</frequency-step-size>
        </properties>
    </flex-client-outbound-queue-processor>
</flex-client>
...
```

You can also configure an outbound queue processor for a particular endpoint in a channel definition. Such an endpoint-specific configuration overrides the global one in the `services-config.xml` file.

Data throttling policies

Data throttling policies determine what happens to Message Service and Data Management Service messages that exceed the specified maximum message limits.

Destinations can use the following throttling policies:

policy	Description
NONE	No data throttling is performed when the maximum message frequency is reached. This setting is equivalent to setting the maximum frequency to 0 (zero).
ERROR	Applies to client-to-server (inbound) messages only. When the maximum frequency limit is exceeded, the server drops the message and sends an error to the client.

policy	Description
IGNORE	When the maximum message frequency is exceeded, the server drops the message but no error is sent to the client.
BUFFER	Advanced feature not available in BlazeDS. See “ Advanced data throttling ” on page 109.
CONFLATE	Advanced feature not available in BlazeDS. See “ Advanced data throttling ” on page 109.

Deserialization validators

You can validate AMF and AMFX deserialization of client-to-server messages when you create an instance of a Java class and set the values of the object’s properties. Validation lets you ensure that the server creates only specific types of objects and that only specific values are allowed on an object’s properties.

Using deserialization validators

Deserialization validator classes must implement the `flex.messaging.validators.DeserializationValidator` interface, which contains the following methods:

- `boolean validateCreation(Class<?> c)`, which validates the creation of a class.
- `boolean validateAssignment(Object instance, int index, Object value)`, which validates the assignment of a value to an index of an Array or List instance.
- `boolean validateAssignment(Object instance, String propertyName, Object value)`, which validates the assignment of a property of an instance of a class to a value.

For more information about the `DeserializationValidator` interface, see the LiveCycle Data Services Javadoc documentation.

Note: If a deserialization validator maintains internal state, guard access and modification of that state with a lock. There is a single deserialization validator per message broker and multiple threads could access the same validator.

You can assign deserialization validators at the top level of the `services-config.xml` file, as the following example shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  ...
    <validators>
      <validator class="mycompany.myvalidators.TestDeserializationValidator"/>
    </validators>
  ...
</services-config>
```

You can also use the runtime configuration feature to dynamically create deserialization validators in a bootstrap service class at server startup. Create a `MessageBroker` and a validator instance in a Java class, and then call the `MessageBroker’s setDeserializationValidator()` method to use the validator. The following example shows this approach:

```
...
MessageBroker broker = MessageBroker.get MessageBroker(null);
TestDeserializationValidator validator = new TestDeserializationValidator();
validator.
// Next, you can call specific add/remove methods on the validator
// for allowed/disallowed classes.
...
broker.setDeserializationValidator(validator);
...
```

The default validator, `flex.messaging.validators.ClassDeserializationValidator`, lets you explicitly allow or disallow specific class types. For more information about this class, see the LiveCycle Data Services Javadoc documentation.

To use the `ClassDeserializationValidator`, you specify disallowed and allowed classes within `disallow-classes` and `allow-classes` elements as the following example shows. You can use fully qualified class names or wildcard expressions in class name values.

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
...
<validators>
    <validator class="flex.messaging.validators.ClassDeserializationValidator">
        <properties>
            <disallow-classes>
            </disallow-classes>
            <allow-classes>
                <class name="java.*"/>
                <class name="\[Ljava.*"/>
                <class name="flex.*"/>
                <class name="\[B*"/>
            </allow-classes>
        </properties>
    </validator>
</validators>
...
</services-config>
```

To help determine which class types are deserialized on the server, you can use the sample `features.validators.serialization.ClassLoggingDeserializationValidator` class. Registering this validator in a development environment lets you capture to the server log all required types that the application uses. When a class is encountered for the first time, an info-level log message that lists the class name is printed to the `Endpoint.Deserialization.Creation` log category. By default, log messages are sent to the console. You can use this information to configure a `ClassDeserializationValidator` instance that disallows creation of all non-required types in the production environment.

The source code for the `ClassLoggingDeserialization` class is in the `resources/samples/validators` directory of your LiveCycle Data Services installation. Unlike the `ClassSerializationValidator` validator, the configuration for this validator does not use any properties. You only need to specify the class name of the validator in a `validator` element.

For more information about the `DeserializationValidator` interface, see the LiveCycle Data Services Javadoc documentation.

More Help topics

[“Run-time configuration” on page 411](#)

Advanced data tuning

LiveCycle Data Services provides a set of advanced data tuning features that give you greater control over the server's data delivery capabilities.

Note: Advanced data tuning features are not available in BlazeDS.

Advanced data tuning includes the following features:

Feature	Description
Reliable messaging	Improves the quality of message delivery between Flex clients and the LiveCycle Data Services server. For more information, see " Reliable messaging " on page 107.
Advanced data throttling	Includes adaptive throttling and the BUFFER and CONFLATE throttling policies. For more information, see " Advanced data throttling " on page 109.
Message priority	Lets you set the priority of server-to-client messages when using the Message Service or Data Management Service. For more information, see " Message priority " on page 111.
Message filtering	Lets you pre-process incoming messages and post-process reply messages and pushed messages. For more information, see " Message filtering " on page 112.

Advanced messaging support

To use most advanced data tuning features, you enable advanced messaging support on the server. The AdvancedMessagingSupport service enables reliable messaging support as the server starts up and handles interactions with client-side AdvancedChannelSet objects at runtime.

The following example shows the text you must add to the services section of the services-config.xml file:

```
...
<services>
...
    <service id="AdvancedMessagingSupport"
        class="flex.messaging.services.AdvancedMessagingSupport"/>
...
</services>
...
```

Suppose you have MXML applications in which you create ChannelSet objects manually and you want to use the listed advanced data tuning features. In this case, modify your applications to use the AdvancedChannelSet class in the fds.swc file rather than the base ChannelSet class in the rpc.swc file.

The following example shows the difference between using the standard ChannelSet object and the AdvancedChannelSet object:

```
...
// Existing base ChannelSet.
var channelSet:ChannelSet = new ChannelSet();
...
// New AdvancedChannelSet.
var channelSet:ChannelSet = new AdvancedChannelSet();
...
```

Reliable messaging

The reliable messaging feature improves the quality of message delivery between clients and destinations on the LiveCycle Data Services server.

Reliable messaging provides the following benefits above what is provided with standard messaging:

Benefit	Description
No message loss	No messages are lost when transient network failures occur if the AdvancedChannelSet object's immediate reliable reconnect attempt succeeds. If this single attempt fails the reliable sequence is closed and the client proceed to regular channel failover and channel set hunting.
Ordered message delivery	Reliable messaging ensures the delivery of correctly ordered messages when using AMF or HTTP channels/endpoints that use more than one TCP socket concurrently to send or receive messages. Message order is ensured even if the intended order differs from the order in which the messages are received from the network. When you use RTMP channels/endpoints, messages are delivered in the correct order whether you use reliable messaging.
No duplicate message processing	There is no duplicate processing of messages even when messages for reliable destination are automatically sent following a disconnect and reconnect.
Reconnection to server following unexpected connection loss	For deployments where network connections are forced to close periodically, reliable messaging allows the client to seamlessly reconnect to the same server. The client resumes exactly where it left off, as long as the server has not crashed, or exited. A single reliable reconnection is attempted immediately following connection loss. This reconnection attempt allows the reliable connection to survive due to an idle timeout or when firewalls, proxies, or other network components force a TCP connection to close periodically. If this single attempt fails, the reliable sequence is closed and the client proceeds to regular channel failover and channel set hunting. The feature currently does not support repeated reliable reconnect attempts.

Note: To use reliable messaging, you enable advanced messaging support.

To enable reliable messaging, you set the `reliable` property to `true` in the network properties section of the destination definition, as the following example shows:

```
...
<destination id="trade-settlement">
    <properties>
        <network>
            <reliable>true</reliable>
        </network>
    </properties>
</destination>
...
```

By default, if a channel was successfully connected before experiencing a fault or disconnection, it attempts a pinned reconnection to the same endpoint URL once. If this immediate reconnection attempt fails, the channel falls back to its previous failover strategy and attempts to fail over to other server nodes in the cluster or fall back to alternate channel protocols. To support IP mobility use cases, the AdvancedChannelSet object provides the following property that you can assign in ActionScript code to extend the time duration for reliable reconnect attempts to continue. One such use case is undocking a laptop and waiting several seconds for its wireless modem to start up and acquire network connectivity and a new IP address.

```
advancedChannelSet.reliableReconnectDuration = 60000; // In milliseconds.
```

The default value is 0. The default behavior is to try a single, immediate reliable reconnection. By assigning a value larger than 0, you give the AdvancedChannelSet more time to attempt to reconnect reliably more than once.

You can also set this value statically in the services-config.xml file within the `flex-client` section, as the following example shows:

```
...
<flex-client>
<reliable-reconnect-duration-millis>60000</reliable-reconnect-duration-millis>
<!-- Idle timeout for FlexClient state at the server, including reliable messaging sequences.
In order to support reliable reconnects consistently across all supported channels and
endpoints, this value must be defined and greater than 0.
Note: Any active sessions/connections keep idle FlexClient instances alive. This timeout
only applies to instances that have no currently active associated sessions/connections.
--&gt;
&lt;timeout-minutes&gt;1&lt;/timeout-minutes&gt;
&lt;/flex-client&gt;
...</pre>
```

Note: If you do not define an explicit FlexClient timeout and at least one RTMP endpoint is registered with the message broker, the server uses a default timeout of 5 seconds for FlexClient timeout. This allows reliable messaging over RTMP to work with no configuration other than setting destinations as reliable. If you want to use a longer reliable reconnect interval, use the optional `timeout-minutes` and `reliable-reconnect-duration-millis` elements within the `flex-client` section of `services-config.xml` file.

Standard behavior without reliable messaging

RTMP channels use a single TCP connection and never deliver messages to the server out of order whether you enable reliable messaging. However, standard AMF channels do not guarantee in-order delivery of sent messages. Server-to-client AMF messages are rolled up into a batch and sent as binary body content in an HTTP POST when Flash Player is able to issue a request. Within a batch, AMF messages are guaranteed to be delivered for processing on the server in order. However, there is a slim chance that clients on slow networks could put themselves into a conflict state unintentionally due to separate HTTP requests that contain individual batches of AMF messages arriving at the server out of order. Nothing in Flash Player or in standard AMF channels prevents sending more than one HTTP request containing a batch of AMF messages at a time. Standard HTTP channels use a unique URLLoader for each message sent, but they serialize the flow of sent messages and only send a subsequent message after an ack (acknowledgement) or fault for the previous message is received. These channels lack the efficiency introduced by the batching behavior of the AMF channels, but they do guarantee in-order message delivery during regular operation.

RTMP channels use a single TCP connection and never receive messages from the server out of order whether you enable reliable messaging. However, standard AMF and HTTP channels sometimes issue concurrent HTTP requests over multiple TCP connections. Polling or streaming, which simulates a duplex connection by overlapping of concurrent HTTP requests more of the time sometimes exacerbates this situation. Standard AMF and HTTP channels provide no guarantee for delivery order across reply messages and pushed messages because data arrives at the client over separate connections concurrently.

Received messages are either replies (acknowledgements/results/faults) for sent messages or they are pushed messages generated by other clients, backend processes, or the server directly. All reliable messages traveling from the server to the client's AdvancedChannelSet object belong to a common sequence. As messages are received, the AdvancedChannelSet object uses a reliable receive buffer to re-order them if necessary, and any in-order messages are removed from the buffer and dispatched to application code. Reply messages are dispatched to the acknowledgment or fault handler of the component that sent the correlated request message. Pushed messages are dispatched to any listeners that have registered for MessageEvents.

Regardless of the runtime message ordering behavior for the various standard channels discussed here, none of these channels can seamlessly survive a transient network disconnect event on its own. By enabling reliable messaging, the channel and server endpoint enforce a guarantee of in-order, once-and-only-once message delivery between the client and server that survives any transient disconnect followed by a successful reconnect to the same server.

Important: *There is a trade-off between quickly detecting that clients have disconnected versus supporting long reliable reconnect durations. For applications where timely server detection of clients disconnecting is important, leave the reliable reconnect duration at its default setting or use a short duration value based on the requirements of your application. For applications where you want to favor reliable reconnections, you can tune this value higher.*

Advanced data throttling

For server-to-client messages, you can use adaptive throttling, which changes message frequency limits based on the actual message processing rates of clients. You can also set policies that determine what happens to messages that exceed the specified maximum message limits.

Note: To use advanced throttling features, you must enable advanced messaging support.

Adaptive client message frequency

When you enable adaptive client message frequency, the server automatically adjusts the frequency of messages sent from server to client while taking the client's actual message processing rate into account. You specify the maximum number of messages per second that the frequency is incrementally increased or decreased.

Add something about fact that the calculations the server does are different depending the type of endpoint you are using. For example, RTMP versus long polling AMF.

To use adaptive client message frequency, you configure the `adaptive-frequency` property of the `flex-client-outbound-queue-processor` element in the `flex-client` section of the `services-config.xml`, as follows:

- 1 Set the `flex-client-queue-processorclass` value to
`"flex.messaging.client.AdvancedOutboundQueueProcessor".`
- 2 Set the value of the `adaptive-frequency` property to `true`.
- 3 You can set the value of the `frequency-step-size` property to the number of messages per second that you want to incrementally increase the message frequency, depending on the server's calculation. This setting is useful when you want to gradually increase the number of messages sent per second.

The following example shows an advanced outbound queue processor configured for adaptive message frequency:

```
...
<flex-client>
    <flex-client-outbound-queue-processor
        class="flex.messaging.client.AdvancedOutboundQueueProcessor">
        <properties>
            <adaptive-frequency>true</adaptive-frequency>
            <frequency-step-size>10</frequency-step-size>
        </properties>
    </flex-client-outbound-queue-processor>
</flex-client>
...
```

You can also configure an outbound queue processor for a particular endpoint in a channel definition. Such an endpoint-specific configuration overrides the global one in the `services-config.xml` file.

Advanced data throttling policies

Destinations can use the following advanced throttling policies. For information about standard throttling policies, see “[Data throttling policies](#)” on page 103.

policy	Description
BUFFER	<p>Buffers messages to the client without dropping or conflating them and without pushing more to the client than the maximum frequency limit. The buffered messages are processed before any newer messages.</p> <p>You can use BUFFER in combination with a maximum message queue size to limit the number of buffered messages that a client can have in its outbound queue. For more information, see “Maximum outbound message queue size” on page 110.</p>
CONFLATE	<p>Applies to the Data Management Service only.</p> <p>Merges messages to the client when the maximum frequency limit is exceeded. This is an advanced BUFFER policy, in which messages are added to a buffer queue but while they are in the queue, they will be merged if possible.</p> <p>You can use CONFLATE in combination with a maximum message queue size to limit the number of buffered messages that a client can have in its outbound queue. For more information, see “Maximum outbound message queue size” on page 110.</p> <p>The CONFLATE policy applies to the following two scenarios:</p> <ul style="list-style-type: none">• Conflation merges DataMessage.UPDATE_OPERATION messages for a single item. These are operations generated when properties of an item are updated. For example, one message might contain an update to the <code>firstname</code> property of an item while another message contains an update to the <code>lastname</code> property. In this case, both changes are merged into a single message. Another example is several updates to the <code>stockprice</code> property, the update message are merged to one and only the latest price is sent, saving unnecessary network traffic and application overhead.• Conflation cancels out a DataMessage.CREATE_OPERATION message followed by a DataMessage.DELETE_OPERATION. This avoids sending two messages unnecessarily.

Maximum outbound message queue size

You can configure the maximum number of messages that a Flex client can have in its outbound queue on the server. The default value of -1 indicates there is no limit on the size of the outbound queue. When the queue is full, the `maxQueueSizeReached()` method of the `AdvancedOutboundQueueProcessor` class is called and the default implementation unsubscribes the client.

To use a maximum message queue size, you configure the `max-queue-size` property of the `flex-client-outbound-queue-processor` element in the `flex-client` section of the `services-config.xml`, as follows:

- 1 Set the `flex-client-queue-processorclass` value to
`"flex.messaging.client.AdvancedOutboundQueueProcessor".`
- 2 Set the value of the `max-queue-size` property to the maximum number of messages to store in the queue.

The following example shows an advanced outbound queue processor configured for adaptive message frequency and maximum outbound message queue size:

```
...
<flex-client>
    <flex-client-outbound-queue-processor
        class="flex.messaging.client.AdvancedOutboundQueueProcessor">
        <properties>
            <max-queue-size>50</max-queue-size>
        </properties>
    </flex-client-outbound-queue-processor>
</flex-client>
...
```

You can also configure an outbound queue processor for a particular endpoint in a channel definition. Such an endpoint-specific configuration overrides the global one in the services-config.xml file.

Message priority

To ensure that messages reach clients in order of importance, you can specify the priority of server-to-client messages when using the Message Service or Data Management Service. Message priority is based on a scale from 0 (zero) to 9, where 0 is the lowest priority and 9 is the highest. The server sends higher priority messages before lower priority messages.

Message prioritization is most useful in situations where messages are queued, such as when you use polling channels or the BUFFER or CONFLATE throttling policy. When you use a polling channel, the server sends the messages that accumulate on the server between polling requests in order of priority. When you use message buffering or conflation, the server sends the messages in the outbound message queue in order of priority.

When you use RTMP or NIO-based HTTP channels, the server sends any messages that accumulate, due to resource constraints or other issues, in order of priority.

By default, messages have a priority of 4. You can override the global default priority by specifying a destination-level default priority. For finer grained control, you can set the default message priority on the client-side Producer or DataService. For still finer grained control, you can set the priority on the priority header of an individual message.

Destination-level message priority

Use destination-level message priority to apply the default priority level to all server-to-client messages sent from a particular destination on the server. This setting overrides the global default priority level.

In the destination definition, set the value of the `message-priority` server property, as the following example shows:

```
...
<destination id="MyTopic">
    <properties>
        <server>
            <message-priority>4</message-priority>
        ...
    </server>
</destination>
```

Producer-level or DataService-level message priority

Use Producer-level or DataService-level message priority to set the default priority to all messages that a Producer or DataService component in a Flex client application create. This setting overrides global and destination-level priority levels.

In ActionScript, set the value of the `Producer.priority` property or `DataService.priority` property, as the following example shows:

```
...
<mx:Script>
    <![CDATA[
        var producer:Producer = new Producer();
        ...
        producer.priority = 2;
        ...
        var ds:DataService = new DataService();
        ...
        ds.priority = 2;
    ]]>
</mx:Script>
...
```

In MXML, set the value of the `priority` attribute of a `Producer` or `DataService` component, as the following example shows:

```
...
<mx:Producer id="producer" destination="chat" priority="2"/>
...
<mx DataService id="ds" destination="meeting" priority="2"/>
...
```

Message-level priority

You can override the default priority of a message at send time by setting the priority header of the message to the desired value. You can set the priority header in ActionScript when a message is created and sent from a `Producer` or `DataService` component on the client to the server. Or, set the priority header in Java when a message is sent directly from the server directly to a client (for example, from a remote object class).

The following example shows a message-level priority header set in ActionScript:

```
...
<mx:Script>
    <![CDATA[
        var msg:AsyncMessage = new AsyncMessage();
        msg.body = "Foo";
        msg.headers[AbstractMessage.PRIORITY_HEADER] = 4;
    ]]>
</mx:Script>
...
```

The following example shows a message-level priority header set in a Java class:

```
...
AsyncMessage msg = new AsyncMessage();
msg.setBody("Foo");
msg.setHeader(Message.PRIORITY_HEADER, 4);
...
```

Message filtering

You use custom message filter classes to pre-process incoming messages and post-process reply messages, and post-process pushed messages that have been flushed from per-client outbound message queues. You configure message filters in the `services-config.xml` file or for run-time configuration in a Java class. You can apply multiple message filters by chaining them together in your configuration.

Message filter classes must subclass either the `flex.messaging.filters.BaseAsyncMessageFilter` class or the `flex.messaging.filters.BaseSyncMessageFilter`. Subclass `BaseSyncMessageFilter` when using servlet-based endpoints and `BaseAsyncMessageFilter` when using NIO-based endpoints. To write a filter that works for either type of endpoint, create two separate classes ideally with common code factored out into a helper/utility class that both filter implementations use internally.

As a first step, you would do what the `Base*` classes do. For more information, see the LiveCycle Data Services Javadoc API documentation.

Asynchronous message filter

The following example shows a simple asynchronous message filter:

```
package features.messaging.filters;
import flex.messaging.MessageContext;
import flex.messaging.client.FlexClient;
import flex.messaging.endpoints.Endpoint;
import flex.messaging.filters.BaseAsyncMessageFilter;
import flex.messaging.messages.AsyncMessage;
import flex.messaging.messages.Message;
/**
 * An asynchronous message filter that works with NIO-based endpoints. You can
 * configure this message filter in services-config.xml as follows:
 *
 * <async-message-filters>
 *   <filter id="SampleAsyncFilter" class="features.messaging.filters.SampleAsyncFilter"/>
 * </async-message-filters>
 */
public class SampleAsyncFilter extends BaseAsyncMessageFilter
{
    /**
     * Filter incoming messages that are marked as heartbeat.
     */
    public void in(final MessageContext context)
    {
        Message message = context.getRequestMessage();
        if (message instanceof HeartbeatMessageExt)
        {
            context.doOut(getPrev()); // Don't process the message further.
```

```
        return;
    }
    super.in(context);
}
/***
 * Filter outgoing messages here.
 */
public void out(final MessageContext context)
{
    super.out(context);
}
/***
 * Filter push messages here.
 */
public AsyncMessage filterPush(final AsyncMessage message, final FlexClient recipient,
final Endpoint endpoint)
{
    return super.filterPush(message, recipient, endpoint);
}
```

Synchronous message filter

The following example shows a simple synchronous message filter:

```
package features.messaging.filters;
import flex.messaging.client.FlexClient;
import flex.messaging.endpoints.Endpoint;
import flex.messaging.filters.BaseSyncMessageFilter;
import flex.messaging.filters.SyncMessageFilterContext;
import flex.messaging.messages.AsyncMessage;
import flex.messaging.messages.Message;
/**
 * A synchronous message filter that works with Servlet-based endpoints. You can
 * configure this message filter in services-config.xml as follows:
 *
 * <sync-message-filters>
 *   <filter id="SampleSyncFilter" class="features.messaging.filters.SampleSyncFilter"/>
 * </sync-message-filters>
 */
public class SampleSyncFilter extends BaseSyncMessageFilter
{
    /**
     * Filter incoming messages that are marked as heartbeat.
     */
    public AsyncMessage filterRequest(final Message message, final Endpoint endpoint, final
SyncMessageFilterContext context)
    {
        return (message instanceof HeartbeatMessageExt) ? null : context.filterRequest(message,
endpoint);
    }
    /**
     * Filter push messages here.
     */
    public void filterPush(final AsyncMessage message, final FlexClient recipient, final
Endpoint endpoint, final SyncMessageFilterContext context)
    {
        context.filterPush(message, recipient, endpoint);
    }
}
```

Message filter configuration

You can assign message filters at the top level of the services-config.xml file. The following example shows the configuration for an asynchronous message filter and a synchronous message filter:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
...
<!-- Asynchronous message filters pre-process incoming messages in "in" method
     and post-process reply messages in "out" method, as well as filtering pushed
     messages in "filterPush" method that travel through all asynchronous
     (NIO-based) endpoints.
-->
<async-message-filters>
    <filter id="asyncFilter" class="my.custom.filters.TestAsyncFilter"/>
</async-message-filters>
<!-- Synchronous message filters pre-process incoming messages and post-process
     reply messages and pushed messages that travel through all synchronous
     (Servlet-based) endpoints.
-->
<sync-message-filters>
    <filter id="syncFilter" class="my.custom.filters.TestSyncFilter"/>
</sync-message-filters>
...
</services-config>
```

Message delivery with adaptive polling

The adaptive polling capability lets you write custom logic to control how messages are queued for delivery to Adobe® Flex™ client applications on a per-client basis.

Adaptive polling

The adaptive polling capability provides a per-client outbound message queue API. You can use this API in custom Java code to manage per-client messaging quality of service based on your criteria for determining and driving quality of service. To use this capability, you create a custom queue processor class and register it in a channel definition in the services-config.xml file.

Using a custom queue processor, you can do such things as conflate messages (combine a new message with an existing message in the queue), order messages according to arbitrary priority rules, filter out messages based on arbitrary rules, and manage flushing (sending) messages to the network layer explicitly. You have full control over the delivery rate of messages to clients on a per-client basis, and the ability to define the order and contents of delivered messages on a per-client basis.

Instances of the flex.messaging.client.FlexClient class on the server maintain the state of each client application. You provide adaptive polling for individual client instances by extending the flex.messaging.client.FlexClientOutboundQueueProcessor class. This class provides an API to manage adding messages to an outbound queue and flushing messages in an outbound queue to the network.

When a message arrives at a destination on the server and it matches a specific client subscription, represented by an instance of the flex.messaging.MessageClient class, the message is routed to an instance of FlexClient where it is added to the queue for the channel/endpoint that the MessageClient subscription was created over. An instance of the flex.messaging.MessageClient class represents a specific client subscription. When a message arrives at a destination on the server, and it matches a client subscription, the message is routed to an instance of the FlexClient class. Then the message is added to the queue for the channel/endpoint that the MessageClient subscription was created over.

The default flush behavior depends on the channel/endpoint that you use. If you use polling channels, a flush is attempted when a poll request is received. If you use direct push channels, a flush is attempted after each new message is added to the outbound queue. You can write Java code to return a `flex.messaging.FlushResult` instance that contains the list of messages to hand off to the network layer, which are sent to the client, and specify an optional wait time to delay the next flush.

For polling channels, a next flush wait time results in the client waiting the specified length of time before issuing its next poll request. For direct push channels, until the wait time is over, the addition of new messages to the outbound queue does not trigger an immediate invocation of flush; when the wait time is up, a delayed flush is invoked automatically. This lets you write code to drive adaptive client polling and adaptive writes over direct connections. You could use this functionality to shed load (for example, to cause clients to poll a loaded server less frequently), or to provide tiered message delivery rates on a per-client basis to optimize bandwidth usage (for example, gold customers could get messages immediately, while bronze customers only receive messages once every 5 minutes).

If an outbound queue processor must adjust the rate of outbound message delivery, it can record its own internal statistics to do so. This could include total number of messages delivered, rate of delivery over time, and so forth. Queue processors that only perform conflation or filtering do not require the overhead of collecting statistics.

The `FlexClientOutboundQueueProcessor`, `FlexClient`, `MessageClient`, and `FlushResult` classes are documented in the public LiveCycle Data Services Javadoc API documentation.

Using a custom queue processor

To use a custom queue processor, you must create a queue processor class, compile it, add it to the class path, and then configure it. The examples in this topic are part of the adaptive polling sample application included in the LiveCycle Data Services samples web application.

Creating a custom queue processor

To create a custom queue processor class, you must extend the `FlexClientOutboundQueueProcessor` class. This class is documented in the public LiveCycle Data Services Javadoc API documentation. It provides the methods described in the following table:

Method	Description
<code>initialize(ConfigMap properties)</code>	Initializes a new queue processor instance after it is associated with its corresponding <code>FlexClient</code> , but before any messages are enqueued.
<code>add(List queue, Message message)</code>	Adds the message to the queue at the desired index, conflates it with an existing message, or ignores it entirely.
<code>FlushResult flush(List queue)</code>	Removes messages from the queue to be flushed out over the network. Can contain an optional wait time before the next flush is invoked.
<code>FlushResult flush(MessageClient messageClient, List queue)</code>	Removes messages from the queue for a specific <code>MessageClient</code> subscription. Can contain an optional wait time before the next flush is invoked.

The following example shows the source code for a custom queue processor class that sets the delay time between flushes in its `flush(List outboundQueue)` method.

```
package flex.samples.qos;

import java.util.ArrayList;
import java.util.List;

import flex.messaging.client.FlexClient;
import flex.messaging.client.FlexClientOutboundQueueProcessor;
import flex.messaging.client.FlushResult;
import flex.messaging.config.ConfigMap;
import flex.messaging.MessageClient;

/**
 * Per client queue processor that applies custom quality of
 * service parameters (in this case: delay).
 * Custom quality of services parameters are read from the client FlexClient
 * instance.
 * In this sample, these parameters are set in the FlexClient instance by
 * the client application using the flex.samples.qos.FlexClientConfigService
 * remote object class.
 * This class is used in the per-client-qos-polling-amf channel definition.
 *
 */
public class CustomDelayQueueProcessor extends FlexClientOutboundQueueProcessor
{
/**
 * Used to store the last time this queue was flushed.
 * Starts off with an initial value of the construct time for the
 * instance.
 */
private long lastFlushTime = System.currentTimeMillis();

/**
 * Driven by configuration, this is the configurable delay time between
 * flushes.
 */
private int delayTimeBetweenFlushes;

public CustomDelayQueueProcessor()
{}

/**
 * Sets up the default delay time between flushes. This default is used
 * if a client-specific
 * value has not been set in the FlexClient instance.
 *
 * @param properties A ConfigMap containing any custom initialization
 * properties.
 */
public void initialize(ConfigMap properties)
{
    delayTimeBetweenFlushes = properties.getPropertyAsInt("flush-delay",-1);
    if (delayTimeBetweenFlushes < 0)
        throw new RuntimeException("Flush delay time forDelayedDeliveryQueueProcessor
            must be a positive value.");
}

/**

```

```
* This flush implementation delays flushing messages from the queue
* until 3 seconds have passed since the last flush.
*
* @param outboundQueue The queue of outbound messages.
* @return An object containing the messages that have been removed
* from the outbound queue
* to be written to the network and a wait time for the next flush
* of the outbound queue
* that is the default for the underlying Channel/Endpoint.
*/
public FlushResult flush(List outboundQueue)
{
    int delay = delayTimeBetweenFlushes;
    // Read custom delay from client's FlexClient instance
    System.out.println("****"+getFlexClient());
    FlexClient flexClient = getFlexClient();
    if (flexClient != null)
    {
        Object obj = flexClient.getAttribute("market-data-delay");
        if (obj != null)
        {
            try {
                delay = Integer.parseInt((String) obj);
            } catch (Exception e) {
            }
        }
    }
}

long currentTime = System.currentTimeMillis();
System.out.println("Flush? " + (currentTime - lastFlushTime) + " < " +delay);
if ((currentTime - lastFlushTime) < delay)
{
    // Delaying flush. No messages will be returned at this point
    FlushResult flushResult = new FlushResult();
    // Don't return any messages to flush.
    // And request that the next flush doesn't occur until 3 seconds since the previous.
    flushResult.setNextFlushWaitTimeMillis((int)(delay -
        (currentTime - lastFlushTime)));
    return flushResult;
}
else // OK to flush.
{
    // Flushing. All queued messages will now be returned
    lastFlushTime = currentTime;
    FlushResult flushResult = new FlushResult();
    flushResult.setNextFlushWaitTimeMillis(delay);
    flushResult.setMessages(new ArrayList(outboundQueue));
    outboundQueue.clear();
    return flushResult;
}
}

public FlushResult flush(MessageClient client, List outboundQueue) {
    return super.flush(client, outboundQueue);
}
```

A Flex client application calls the following remote object to set the delay time between flushes on CustomDelayQueueProcessor:

```
package flex.samples.qos;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

import flex.messaging.FlexContext;
import flex.messaging.client.FlexClient;

public class FlexClientConfigService
{

    public void setAttribute(String name, Object value)
    {
        FlexClient flexClient = FlexContext.getFlexClient();
        flexClient.setAttribute(name, value);
    }

    public List getAttributes()
    {
        FlexClient flexClient = FlexContext.getFlexClient();
        List attributes = new ArrayList();
        Enumeration attrNames = flexClient.getAttributeNames();
        while (attrNames.hasMoreElements())
        {
            String attrName = (String) attrNames.nextElement();
            attributes.add(new Attribute(attrName, flexClient.getAttribute(attrName)));
        }

        return attributes;
    }

    public class Attribute {

        private String name;
        private Object value;

        public Attribute(String name, Object value) {
            this.name = name;
            this.value = value;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public Object getValue() {
            return value;
        }
        public void setValue(Object value) {
            this.value = value;
        }
    }
}
```

Configuring a custom queue processor

You register custom implementations of the FlexClientOutboundQueueProcessor class on a per-channel/endpoint basis. To register a custom implementation, you configure a `flex-client-outbound-queue` property in a channel definition in the services-config.xml file, as the following example shows:

```
<channel-definition id="per-client-qos-polling-amf"
class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://localhost:8400/lcds-samples/messagebroker/qosamfpolling"
        class="flex.messaging.endpoints.AMFEEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-millis>500</polling-interval-millis>
        <flex-client-outbound-queue-processor
            class="flex.samples.qos.CustomDelayQueueProcessor">
            <properties>
                <flush-delay>5000</flush-delay>
            </properties>
        </flex-client-outbound-queue-processor>
    </properties>
</channel-definition>
```

This example shows how you can also specify arbitrary properties to be passed into the `initialize()` method of your queue processor class after it has been constructed and has been associated with its corresponding FlexClient instance, but before any messages are enqueued. In this case, the `flush-delay` value is passed into the `initialize()` method. This is the default value that is used if a client does not specify a flush delay value.

You then specify the channel in your message destination, as the bold text in the following example shows:

```
<destination id="market-data-feed">
    <properties>
        <network>
            <subscription-timeout-minutes>0</subscription-timeout-minutes>
        </network>
        <server>
            <message-time-to-live>0</message-time-to-live>
            <durable>true</durable>
            <allow-subtopics>true</allow-subtopics>
            <subtopic-separator>.</subtopic-separator>
        </server>
    </properties>
    <channels>
        <channel ref="per-client-qos-rtmp"/>
    </channels>
</destination>
```

Measuring message processing performance

As part of preparing your application for final deployment, you can test its performance to look for ways to optimize it. One place to examine performance is in the message processing part of the application. To help you gather this performance information, enable the gathering of message timing and sizing data.

About measuring message processing performance

The mechanism for measuring message processing performance is disabled by default. When enabled, information regarding message size, server processing time, and network travel time is available to the client that pushed a message to the server, to a client that received a pushed message from the server, or to a client that received an acknowledge message from the server in response a pushed message. A subset of this information is also available for access on the server.

You can use this mechanism across all channel types, including polling and streaming channels, that communicate with the LiveCycle Data Services server. However, this mechanism does not work when you make a direct connection to an external server by setting the `useProxy` property to `false` for the `HTTPService` and `WebService` tags because it bypasses the LiveCycle Data Services Proxy Server.

The `MessagePerformanceUtils` class defines the available message processing metrics. When a consumer receives a message, or a producer receives an acknowledge message, the consumer or producer extracts the metrics into an instance of the `MessagePerformanceUtils` class, and then accesses the metrics as properties of that class. For a complete list of the available metrics, see “[Available message processing metrics](#)” on page 123.

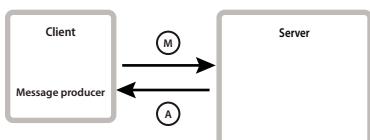
Measuring performance for different channel types

The types of metrics that are available and their calculations, depend on the channel configuration over which a message is sent from or received by the client.

Producer acknowledge scenario

In the producer acknowledge scenario, a producer sends a message to a server over a specific channel. The server then sends an acknowledge message back to the producer.

The following image shows the producer acknowledge scenario:



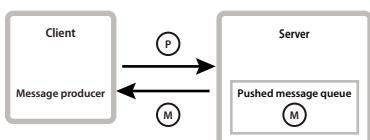
M. Message sent to server. A. Acknowledge message

If you enable the gathering of message processing metrics, the producer adds information to the message before sending it to the server, such as the send time and message size. The server copies the information from the message to the acknowledge message. Then the server adds additional information to the acknowledge message, such as the response message size and server processing time. When the producer receives the acknowledge message, it uses all of the information in the message to calculate the metrics defined by the `MessagePerformanceUtils` class.

Message polling scenario

In a message polling scenario, a consumer polls a message channel to determine if a message is available on the server. On receiving the polling message, the server pushes any available message to the consumer.

The following image shows the polling scenario:



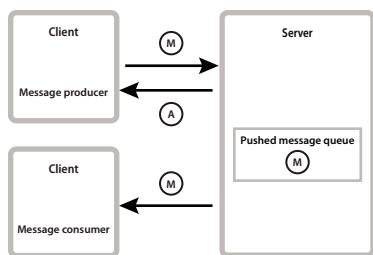
P. Polling message sent. M. Message pushed to client from server.

If you enable the gathering of message processing metrics in this scenario, the consumer obtains performance metrics about the poll-response transaction, such as the response message size and server processing time. The metrics also include information about the message returned by the server. This information lets the consumer determine how long the message was waiting before it was pushed. However, the metric information does not identify the client that originally pushed the message onto the server.

Message streaming scenario

In the streaming scenario, the server pushes a message to a consumer when a message is available; the consumer itself does not initiate the transaction.

The following image shows this scenario:



Message streaming scenario.

In this scenario, the message producer pushes a message, and then receives an acknowledge message. The producer can obtain metric information as described in “[Producer acknowledge scenario](#)” on page 122.

When the server pushes the message to the consumer, the message contains information from the original message from the producer, and the metric information that the server added. The consumer can then examine the metric data, including the time from when the producer pushed the message until the consumer received it.

Measuring message processing performance for streaming and RTMP channels

For streaming and RTMP channels, a pushed message is sent to a consumer without the client first sending a polling message. In this case, the following metrics are not available to the consumer for the pushed message, but instead are set to 0:

- networkRTT
- serverPollDelay
- totalTime

Available message processing metrics

The following table lists the available message processing metrics defined by the `MessagePerformanceUtils` class:

Property	Description
clientReceiveTime	The number of milliseconds since the start of the UNIX epoch, January 1, 1970, 00:00:00 GMT, to when the client received a response message from the server.
messageSize	The size of the original client message, in bytes, as measured during deserialization by the server endpoint.
networkRTT	The duration, in milliseconds, from when a client sent a message to the server until it received a response, excluding the server processing time. This value is calculated as <code>totalTime - serverProcessingTime</code> . If a pushed message is using a streaming or RTMP channel, the metric is meaningless because the client does not initiate the pushed message; the server sends a message to the client whenever a message is available. Therefore, for a message pushed over a streaming or RTMP channel, this value is 0. However, for an acknowledge message sent over a streaming or RTMP channel, the metric contains a valid number.
originatingMessageSentTime	The timestamp, in milliseconds since the start of the UNIX epoch on January 1, 1970, 00:00:00 GMT, to when the client that caused a push message sent its message. Only populated for a pushed message, but not for an acknowledge message.
originatingMessageSize	Size, in bytes, of the message that originally caused this pushed message. Only populated for a pushed message, but not for an acknowledge message.
pushedMessageFlag	Contains <code>true</code> if the message was pushed to the client but is not a response to a message that originated on the client. For example, when the client polls the server for a message, <code>pushedMessageFlag</code> is <code>false</code> . When you are using a streaming channel, <code>pushedMessageFlag</code> is <code>true</code> . For an acknowledge message, <code>pushedMessageFlag</code> is <code>false</code> .
pushOneWayTime	Time, in milliseconds, from when the server pushed the message until the client received it. Note: This value is only relevant if the server and receiving client have synchronized clocks. Only populated for a pushed message, but not for an acknowledge message.
responseMessageSize	The size, in bytes, of the response message sent to the client by the server as measured during serialization at the server endpoint.
serverAdapterExternalTime	Time, in milliseconds, spent in a module invoked from the adapter associated with the destination for this message, before either the response to the message was ready or the message had been prepared to be pushed to the receiving client. This value corresponds to the message processing time on the server.
serverAdapterTime	Processing time, in milliseconds, of the message by the adapter associated with the destination before the response to the message was ready or the message was prepared to be pushed to the receiving client. The processing time corresponds to the time that your code on the server processed the message, not when LiveCycle Data Services processed the message.
serverNonAdapterTime	Server processing time spent outside the adapter associated with the destination of this message. Calculated as <code>serverProcessingTime - serverAdapterTime</code> .
serverPollDelay	Time, in milliseconds, that this message sat on the server after it was ready to be pushed to the client but before it was picked up by a poll request. For a streaming or RTMP channel, this value is always 0.
serverPrePushTime	Time, in milliseconds, between the server receiving the client message and the server beginning to push the message out to other clients.

Property	Description
serverProcessingTime	Time, in milliseconds, between server receiving the client message and either the time the server responded to the received message or has the pushed message ready to be sent to a receiving client. For example, in the producer-acknowledge scenario, this value is the time from when the server receives the message and sends the acknowledge message back to the producer. In a polling scenario, it is the time between the arrival of the polling message from the consumer and any message returned in response to the poll.
serverSendTime	The number of milliseconds since the start of the UNIX epoch, January 1, 1970, 00:00:00 GMT, to when the server sent a response message back to the client.
totalPushTime	Time, in milliseconds, from when the originating client sent a message and the time that the receiving client received the pushed message. Note: This value is only relevant if the two clients have synchronized clocks. Only populated for a pushed message, but not for an acknowledge message.
totalTime	Time, in milliseconds, between this client sending a message and receiving a response from the server. This property contains 0 for a streaming or RTMP channel.

Considerations when measuring message processing performance

The mechanism that measures message processing performance attempts to minimize the overhead required to collect information so that all timing information is as accurate as possible. However, take into account the following considerations when you use this mechanism.

Synchronize the clocks on different computers

The metrics defined by the `MessagePerformanceUtils` class include the `totalPushTime`. The `totalPushTime` is a measure of the time from when the originating message producer sent a message until a consumer receives the message. This value is determined from the timestamp added to the message when the producer sends the message, and the timestamp added to the message when the consumer receives the message. However, to calculate a valid value for the `totalPushTime` metric, the clocks on the message-producing computer and on the message-consuming computer must be synchronized.

Another metric, `pushOneWayTime`, contains the time from when the server pushed the message until the consumer received it. This value is determined from the timestamp added to the message when the server sends the message, and the timestamp added to the message when the consumer receives the message. To calculate a valid value for the `pushOneWayTime` metric, the clocks on the message consuming computer and on the server must be synchronized.

One option is to perform your testing in a lab environment where you can ensure that the clocks on all computers are synchronized. For the `totalPushTime` metric, you can ensure that the clocks for the producer and consumer applications are synchronized by running the applications on the same computer. Or, for the `pushOneWayTime` metric, you can run the consumer application and server on the same computer.

Perform different tests for message timing and sizing

The mechanism for measuring message processing performance lets you enable the tracking of timing information, of sizing information, or both. The gathering of timing-only metrics is minimally intrusive to your overall application performance. The gathering of sizing metrics involves more overhead time than gathering timing information.

Therefore, you can run your tests twice: once for gathering timing information and once for gathering sizing information. In this way, the timing-only test can eliminate any delays caused by calculating message size. You can then combine the information from the two tests to determine your final results.

Measuring message processing performance

The mechanism for measuring message processing performance is disabled by default. When you enable it, you can use the MessagePerformanceUtils class to access the metrics from a message received by a client.

Enabling message processing metrics

You use two parameters in a channel definition to enable message processing metrics:

- <record-message-times>
- <record-message-sizes>

Set these parameters to `true` or `false`; the default value is `false`. You can set the parameters to different values to capture only one type of metric. For example, the following channel definition specifies to capture message timing information, but not message sizing information:

```
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf"
    class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
    <properties>
        <record-message-times>true</record-message-times>
        <record-message-sizes>false</record-message-sizes>
    </properties>
</channel-definition>
```

Using the MessagePerformanceUtils class

The MessagePerformanceUtils class is a client-side class that you use to access the message processing metrics. You create an instance of the MessagePerformanceUtils class from a message pushed to the client by the server or from an acknowledge message.

The following example shows a message producer that uses the acknowledge message to display in a TextArea control the metrics for a message pushed to the server:

```
<?xml version="1.0"?>
<! -- mpi\ChatACK.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[
            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
                producer.send(message);
                msg.text = "";
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```
private function ackHandler(event:MessageEvent):void {
    var mpiutil:MessagePerformanceUtils =
        new MessagePerformanceUtils(event.message);
    myTAAck.text = "totalTime = " + String(mpiutil.totalTime);
    myTAAck.text = myTAAck.text + "\n" + "messageSize= " +
        String(mpiutil.messageSize);
}

]]>
</mx:Script>

<mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>

<mx:Label text="Acknowledge metrics"/>
<mx:TextArea id="myTAAck" width="100%" height="20%"/>

<mx:Panel title="Chat" width="100%" height="100%">
    <mx:TextArea id="log" width="100%" height="100%"/>
    <mx:ControlBar>
        <mx:TextInput id="msg" width="100%" enter="send()"/>
        <mx:Button label="Send" click="send()"/>
    </mx:ControlBar>
</mx:Panel>

</mx:Application>
```

In this example, you write an event handler for the `acknowledge` event to display the metrics. The event handler extracts the metric information from the `acknowledge` message, and then displays the `MessagePerformanceUtils.totalTime` and `MessagePerformanceUtils.messageSize` metrics in a `TextArea` control.

You can also use the `MessagePerformanceUtils.prettyPrint()` method to display the metrics. The `prettyPrint()` method returns a formatted String that contains nonzero and non-null metrics. The following example modifies the event handler for the previous example to use the `prettyPrint()` method:

```
// Event handler to write metrics to the TextArea control.
private function ackHandler(event:MessageEvent):void {
    var mpiutil:MessagePerformanceUtils = new MessagePerformanceUtils(event.message);
    myTAAck.text = mpiutil.prettyPrint();
}
```

The following example shows the output from the `prettyPrint()` method that appears in the `TextArea` control:

```
Original message size(B): 509
Response message size(B): 562
Total time (s): 0.016
Network Roundtrip time (s): 0.016
```

A message consumer can write an event handler for the `message` event to display metrics, as the following example shows:

```
<?xml version="1.0"?>
<!!-- mpi\ChatConsume.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="consumer.subscribe();">

    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
                producer.send(message);
                msg.text = "";
            }

            // Event handler to write metrics to the TextArea control.
            private function ackHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAAck.text = mpiutil.prettyPrint();
            }

            // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();

            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>
    <mx:Consumer id="consumer" destination="chat" message="messageHandler(event)"/>

    <mx:Label text="ack metrics"/>
    <mx:TextArea id="myTAAck" width="100%" height="20%" text="ack"/>

    <mx:Label text="receive metrics"/>
    <mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

    <mx:Panel title="Chat" width="100%" height="100%">
        <mx:TextArea id="log" width="100%" height="100%"/>
        <mx:ControlBar>
            <mx:TextInput id="msg" width="100%" enter="send()"/>
            <mx:Button label="Send" click="send()"/>
        </mx:ControlBar>
    </mx:Panel>

</mx:Application>
```

In this example, you use the `prettyPrint()` method to write the metrics for the received message to a `TextArea` control. The following example shows this output:

```
Response message size(B): 560
PUSHED MESSAGE INFORMATION:
Total push time (s): 0.016
Push one way time (s): 0.016
Originating Message size (B): 509
```

You can gather metrics for `HTTPService` and `WebService` tags when they use the Proxy Service, as defined by setting the `useProxy` property to `true` for the `HTTPService` and `WebService` tags. The following example gathers metrics for an `HTTPService` tag:

```
<?xml version="1.0" encoding="utf-8"?>
<!- mpi\main.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="#FFFFFF">

    <mx:Script>
        <! [CDATA[
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;

            // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();
            }
        ]]>
    </mx:Script>

    <mx:Label text="Message metrics"/>
    <mx:TextArea id="myTAMess" width="100%" height="20%"/>

    <mx:HTTPService id="srv"
        destination="catalog"
        useProxy="true"
        result="messageHandler(event);"/>

    <mx:DataGrid dataProvider="{srv.lastResult.catalog.product}"
        width="100%" height="100%"/>

    <mx:Button label="Get Data" click="srv.send()"/>
</mx:Application>
```

When using LiveCycle Data Services Data Management Service, you can use event handlers on the `DataService` class to handle metrics, as the following example shows:

```
<mx:Script>
<! [CDATA[

    import mx.messaging.events.MessageEvent;
    import mx.messaging.messages.MessagePerformanceUtils;

    // Event handler to write metrics to the TextArea control.
    private function messageHandler(event:MessageEvent):void {
        var mpiutil:MessagePerformanceUtils = new MessagePerformanceUtils(event.message);
        myTAMess.text = mpiutil.prettyPrint();
    }
]]>
</mx:Script>

<mx:DataService id="ds" destination="inventory" result="messageHandler(event);"/>

<mx:Label text="receive metrics"/>
<mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

<mx:Button label="Get Data" click="ds.fill(products)"/>
```

Using the server-side classes to gather metrics

For managed endpoints, you can access the total number of bytes serialized and deserialized by using the following methods of the `flex.management.runtime.messaging.endpoints.EndpointControlMBean` interface:

- `getBytesDeserialized()`
Returns the total number of bytes deserialized by this endpoint during its lifetime.
- `getBytesSerialized()`
Returns the total number of bytes serialized by this endpoint during its lifetime.

The `flex.management.runtime.messaging.endpoints.EndpointControlMBean` class implements these methods.

Writing messaging metrics to the log files

You can write messaging metrics to the client-side log file if you enable the metrics. To enable the metrics, set the `<record-message-times>` or `<record-mssage-sizes>` parameter to `true`, and the client-side log level to `DEBUG`. Messages are written to the log when a client receives an acknowledgment for a pushed message, or a client receives a pushed message from the server. The metric information appears immediately following the debug information for the received message.

The following example initializes logging and sets the log level to `DEBUG`:

```
<?xml version="1.0"?>
<!-- mpi\ChatEverything.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="consumer.subscribe();initLogging();">

    <mx:Script>
        <![CDATA[

            import mx.messaging.messages.AsyncMessage;
            import mx.messaging.messages.IMessage;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.MessagePerformanceUtils;
            import mx.controls.Alert;
            import mx.collections.ArrayCollection;
            import mx.logging.targets.*;
            import mx.logging.*;

            // Event handler to send the message to the server.
            private function send():void
            {
                var message:IMessage = new AsyncMessage();
                message.body.chatMessage = msg.text;
                producer.send(message);
                msg.text = "";
            }

            // Event handler to write metrics to the TextArea control.
            private function ackHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAAck.text = mpiutil.prettyPrint();
            }

            // Event handler to write metrics to the TextArea control for the message consumer.
            private function messageHandler(event:MessageEvent):void {
                var mpiutil:MessagePerformanceUtils =
                    new MessagePerformanceUtils(event.message);
                myTAMess.text = mpiutil.prettyPrint();
            }

            // Initialize logging and set the log level to DEBUG.
            private function initLogging():void {
                // Create a target.
                var logTarget:TraceTarget = new TraceTarget();

                // Log all log levels.
                logTarget.level = LogEventLevel.DEBUG;

                // Add date, time, category, and log level to the output.
                logTarget.includeDate = true;
                logTarget.includeTime = true;
                logTarget.includeCategory = true;
                logTarget.includeLevel = true;

                // Begin logging.
                Log.addTarget(logTarget);
            }
        ]]>
    

```

```
]]>
</mx:Script>

<mx:Producer id="producer" destination="chat" acknowledge="ackHandler(event)"/>
<mx:Consumer id="consumer" destination="chat" message="messageHandler(event)"/>

<mx:Label text="ack metrics"/>
<mx:TextArea id="myTAAck" width="100%" height="20%" text="ack"/>

<mx:Label text="receive metrics"/>
<mx:TextArea id="myTAMess" width="100%" height="20%" text="rec"/>

<mx:Panel title="Chat" width="100%" height="100%">
    <mx:TextArea id="log" width="100%" height="100%"/>
    <mx:ControlBar>
        <mx:TextInput id="msg" width="100%" enter="send()"/>
        <mx:Button label="Send" click="send()"/>
    </mx:ControlBar>
</mx:Panel>
</mx:Application>
```

For more information on logging, see the Flex documentation set.

By default, on Microsoft Windows the log file is written to the file C:\Documents and Settings\USERNAME\Application Data\Macromedia\Flash Player\Logs\flashlog.txt. The following excerpt is from the log file for the message "My test message":

```
2/14/2008 11:20:18.806 [DEBUG] mx.messaging.Channel 'my-rtmp' channel got connect attempt
status. (Object)#0
    code = "NetConnection.Connect.Success"
    description = "Connection succeeded."
    details = (null)
    DSMessagingVersion = 1
    id = "D46A822C-962B-4651-6F2A-DCB41130C4CF"
    level = "status"
    objectEncoding = 3
2/14/2008 11:20:18.837 [INFO] mx.messaging.Channel 'my-rtmp' channel is connected.
2/14/2008 11:20:18.837 [DEBUG] mx.messaging.Channel 'my-rtmp' channel sending message:
(mx.messaging.messages::CommandMessage)
    body=(Object)#0
    clientId=(null)
    correlationId=""
    destination="chat"
    headers=(Object)#0
    messageId="E2F6B35E-42CD-F088-4B7A-18BF1515F142"
    operation="subscribe"
    timeToLive=0
    timestamp=0
2/14/2008 11:20:18.868 [INFO] mx.messaging.Consumer 'consumer' consumer connected.
2/14/2008 11:20:18.868 [INFO] mx.messaging.Consumer 'consumer' consumer acknowledge for
subscribe. Client id 'D46A82C3-F419-0EBF-E2C8-330F83036D38' new timestamp 1203006018867
2/14/2008 11:20:18.884 [INFO] mx.messaging.Consumer 'consumer' consumer acknowledge of
'E2F6B35E-42CD-F088-4B7A-18BF1515F142'.
2/14/2008 11:20:18.884 [DEBUG] mx.messaging.Consumer Original message size(B): 626
Response message size(B): 562

2/14/2008 11:20:25.446 [INFO] mx.messaging.Producer 'producer' producer sending message
```

```
'CF89F532-A13D-888D-D929-18BF2EE61945'  
2/14/2008 11:20:25.462 [INFO] mx.messaging.Producer 'producer' producer connected.  
2/14/2008 11:20:25.477 [DEBUG] mx.messaging.Channel 'my-rtmp' channel sending message:  
(mx.messaging.messages::AsyncMessage)#0  
    body = (Object)#1  
        chatMessage = "My test message"  
        clientId = (null)  
        correlationId = ""  
        destination = "chat"  
        headers = (Object)#2  
        messageId = "CF89F532-A13D-888D-D929-18BF2EE61945"  
        timestamp = 0  
        timeToLive = 0  
2/14/2008 11:20:25.571 [DEBUG] mx.messaging.Channel 'my-rtmp' channel got message  
(mx.messaging.messages::AsyncMessageExt)#0  
    body = (Object)#1  
        chatMessage = "My test message"  
        clientId = "D46A82C3-F419-0EBF-E2C8-330F83036D38"  
        correlationId = ""  
        destination = "chat"  
        headers = (Object)#2  
        DSMPIO = (mx.messaging.messages::MessagePerformanceInfo)#3  
            infoType = "OUT"  
            messageSize = 575  
            overheadTime = 0  
            pushedFlag = true  
            receiveTime = 1203006025556  
            recordMessageSizes = false  
            recordMessageTimes = false  
            sendTime = 1203006025556  
            serverPostAdapterExternalTime = 0  
            serverPostAdapterTime = 0  
            serverPreAdapterExternalTime = 0  
            serverPreAdapterTime = 0  
            serverPrePushTime = 0  
        DSMPIP = (mx.messaging.messages::MessagePerformanceInfo)#4  
            infoType = (null)  
            messageSize = 506  
            overheadTime = 0  
            pushedFlag = false  
            receiveTime = 1203006025556  
            recordMessageSizes = true  
            recordMessageTimes = true
```

```
sendTime = 1203006025556
serverPostAdapterExternalTime = 1203006025556
serverPostAdapterTime = 1203006025556
serverPreAdapterExternalTime = 0
serverPreAdapterTime = 1203006025556
serverPrePushTime = 1203006025556
messageId = "CF89F532-A13D-888D-D929-18BF2EE61945"
timestamp = 1203006025556
timeToLive = 0

2/14/2008 11:20:25.696 [DEBUG] mx.messaging.Channel Response message size(B): 575
PUSHED MESSAGE INFORMATION:
Originating Message size (B): 506

2/14/2008 11:20:25.712 [INFO] mx.messaging.Producer 'producer' producer acknowledge of
'CF89F532-A13D-888D-D929-18BF2EE61945'.
2/14/2008 11:20:25.712 [DEBUG] mx.messaging.Producer Original message size(B): 506
Response message size(B): 562
Total time (s): 0.156
Network Roundtrip time (s): 0.156
```

Chapter 4: RPC services

Using RPC services

Remote Procedure Call (RPC) components let a client application make calls to operations and services across a network. The three RPC components are the `RemoteObject`, `HTTPService`, and `WebService` components. Your Flex client code uses these components to access remote object services, web services, and HTTP services.

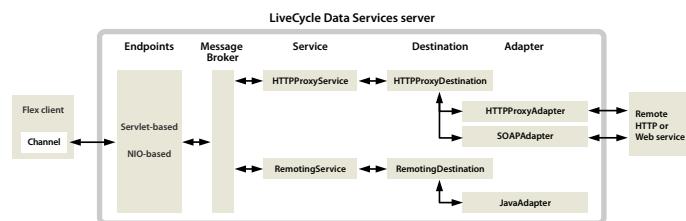
When you use `HTTPService` and `WebService` components with LiveCycle Data Services or BlazeDS, you go through the Proxy Service on the server, which proxies your requests so you do not need a cross domain policy file. When you use the Proxy Service, you can also secure access to services. When you use `RemoteObject` components with LiveCycle Data Services or BlazeDS, you go through the Remoting Service, which exposes plain old Java objects (POJOs) as service destinations that your client-side `RemoteObject` components can access on the server.

Client-side RPC components

The RPC components are designed for client applications in which a call and response model is a good choice for accessing external data. These components let the client make asynchronous requests to remote services that process the requests, and then return data to your Flex application.

The RPC components call a remote service, and then store response data from the service in an ActionScript or XML object from which you obtain the data. You can use the RPC components in the client application to work with three types of RPC services: remote object services with the `RemoteObject` component, web services with the `WebService` component, and HTTP services with the `HTTPService` component.

When you use LiveCycle Data Services or BlazeDS, the client typically contacts a destination, which is an RPC service that has a corresponding server-side configuration. The following diagram shows how the RPC components in a Flex client application interact with LiveCycle Data Services services:



Server-side services, destinations, and adapters

The server-side services, destinations, and adapters that you use depend on the RPC component, as the following table shows:

Component	Service	Destination	Adapter
HTTPService	HTTPProxyService	HTTPProxyDestination	HTTPProxyAdapter
WebService	HTTPProxyService	HTTPProxyDestination	SOAPAdapter
RemoteObject	RemotingService	RemotingDestination	JavaAdapter

RPC channels

With RPC services, you often use an AMFChannel on the client that calls a destination (endpoint) on the server. The AMFChannel uses binary AMF encoding over HTTP. If binary data is not allowed, then you can use an HTTPChannel, which is AMFX (AMF in XML) over HTTP. For more information on channels, see “[Client and server architecture](#)” on page 27.

Types of client-side RPC components

Use RPC components to add enterprise functionality, such as proxying of service traffic from different domains, client authentication, whitelists of permitted RPC service URLs, security, server-side logging, localization support, and centralized management of RPC services.

Note: You can use the `HTTPService` and `WebService` components to call HTTP services or web services directly, without going through the server-side proxy service. For more information, see “[Using `HTTPService` and `WebService` without a destination](#)” on page 145.

By default, Adobe Flash Player blocks access to any host that is not exactly equal to the one used to load an application. Therefore, if you do not use LiveCycle Data Services to proxy requests, an HTTP or web service must either be on the server hosting your application, or the remote server that hosts the HTTP or web service must define a `crossdomain.xml` file. A `crossdomain.xml` file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. The `crossdomain.xml` file must be in the web root of the server that the Flex application is contacting.

Use `RemoteObject` components to access remote Java objects on the LiveCycle Data Services server without configuring them as SOAP-compliant web services. You cannot use `RemoteObject` components without LiveCycle Data Services or ColdFusion.

HTTPService component

`HTTPService` components let you send HTTP GET, POST, HEAD, OPTIONS, PUT, TRACE or DELETE requests, and include data from HTTP responses in a Flex application. Flex does not support multipart form POST requests. However, when you do not go through the `HTTPProxyService`, you can use only HTTP GET or POST methods.

An HTTP service can be any HTTP URI that accepts HTTP requests and sends responses. Another common name for this type of service is a REST-style web service. REST stands for Representational State Transfer and is an architectural style for distributed hypermedia systems. For more information about REST, see www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

`HTTPService` components are a good option when you cannot expose the same functionality as a SOAP web service or a remote object service. For example, you can use `HTTPService` components to interact with JavaServer Pages (JSPs), servlets, and ASP pages that are not available as web services or Remoting Service destinations.

Use an `HTTPService` component for CGI-like interaction in which you use HTTP GET, POST, HEAD, OPTIONS, PUT, TRACE, or DELETE to send a request to a specified URI. When you call the `HTTPService` object's `send()` method, it makes an HTTP request to the specified URI, and an HTTP response is returned. Optionally, you can pass arguments to the specified URI.

WebService component

`WebService` components let you access *web services*, which are software modules with methods. Web service methods are commonly referred to as *operations*. Web service interfaces are defined by using XML. Web services provide a standards-compliant way for software modules that are running on a variety of platforms to interact with each other. For more information about web services, see the web services section of the World Wide Web Consortium website at www.w3.org/2002/ws/.

Flex applications can interact with web services that define their interfaces in a Web Services Description Language (WSDL) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages.

Flex supports WSDL 1.1, which is described at www.w3.org/TR/wsdl. Flex supports both RPC-encoded and document-literal web services.

Flex applications support web service requests and results that are formatted as SOAP messages and are transported over HTTP. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

You can use a `WebService` component to connect to a SOAP-compliant web service when web services are an established standard in your environment. `WebService` components are also useful for objects that are within an enterprise environment, but not necessarily available on the sourcepath of the Flex web application.

RemoteObject component

`RemoteObject` components let you access the methods of server-side Java objects, without manually configuring the objects as web services. You can use `RemoteObject` components in MXML or ActionScript.

You can use `RemoteObject` components with a stand-alone LiveCycle Data Services web application or ColdFusion. When using a LiveCycle Data Services web application, you configure the objects that you want to access as Remoting Service destinations in a LiveCycle Data Services configuration file or by using LiveCycle Data Services run-time configuration. For information on using `RemoteObject` components with ColdFusion, see the ColdFusion documentation.

Use a `RemoteObject` component instead of a `WebService` component when objects are not already published as web services, web services are not used in your environment, or you would rather use Java objects than web services. You can use a `RemoteObject` component to connect to a local Java object that is in the LiveCycle Data Services or ColdFusion web application sourcepath.

When you use a `RemoteObject` tag, data is passed between your application and the server-side object in the binary Action Message Format (AMF) format.

Using an RPC component

The following example shows MXML code for a `WebService` component. This example connects to a LiveCycle Data Services destination, calls the `getProducts()` operations of the web service in response to the `click` event of a `Button` control, and displays the result data in a `DataGrid` control.

```
<?xml version="1.0"?>
<!- ds\rpc\RPCIntroExample.mxml -->
&ltmx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[

            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the DataGrid control.
                // The operation returns an Array containing product ID, name, and price.
                myDG.dataProvider=event.result;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!- Define a WebService component and connect to a service destination. -->
    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!- Call the getProducts() operation of the web service.
    The operation takes no parameters. -->
    <mx:Button label="Get Data" click="adbe_news.getProducts();"/>

    <!- Define a DataGrid control to display the results of the web service. -->
    <mx:DataGrid id="myDG" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="productId" headerText="Product Id"/>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="price" headerText="Price"/>
        </mx:columns>
    </mx:DataGrid>

</mx:Application>
```

This example shows the basic process for using any RPC control, including the following:

- Defines the component in MXML. You can also define a component in ActionScript. For more information, see “[HTTP services and web services](#)” on page 140.
- Specifies the LiveCycle Data Services destination that the RPC component connects to. For more information, see “[Using destinations](#)” on page 140.
- Invokes the `getProducts()` operation in response to a Button `click` event. In this example, the `getProducts()` operation takes no parameters. For more information on parameter passing, see “[Passing parameters to a service](#)” on page 149.

- Defines event handlers for the `result` event and for the `fault` event. Calls to an RPC service are asynchronous. After you invoke an asynchronous call, your application does not block execution and wait for immediate response, but continues to execute. Components use events to signal that the service has completed. In this example, the handler for the `result` event populates the DataGrid control with the results of the operation. For more information, see “[Handling service events](#)” on page 171.

RPC components versus other technologies

The way that Flex works with data sources and data is different from other web application environments, such as JSP, ASP, and ColdFusion.

Client-side processing and server-side processing

Unlike a set of HTML templates created using JSPs and servlets, ASP, or CFML, the files in a Flex application are compiled into a binary SWF file that is sent to the client. When a Flex application makes a request to an external service, the SWF file is not recompiled and no page refresh is required.

The following example shows MXML code for calling a web service. When a user clicks the Button control, client-side code calls the web service, and result data is returned into the binary SWF file without a page refresh. The result data is then available to use as dynamic content within the application.

```
<?xml version="1.0"?>
<!!-- ds\rpc\RPCIntroExample2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <!-- Declare a WebService component (the specified WSDL URL is not functional). -->
    <mx:WebService id="WeatherService"
        destination="wsDest"/>

    <mx:Button label="Get Weather"
        click="WeatherService.GetWeather(input.text);"/>

    <mx:TextInput id="input"/>
</mx:Application>
```

The following example shows JSP code for calling a web service using a JSP custom tag. When a user requests this JSP, the web service request is made on the server instead of on the client, and the result is used to generate content in the HTML page. The application server regenerates the entire HTML page before sending it back to the browser.

```
<%@ taglib prefix="web" uri="webservicetag" %>

<% String str1="BRL";
String str2="USD";%>

<!-- Call the web service. -->
<web:invoke
    url="http://www.itfinity.net:8008/soap/exrates/default.asp"
    namespace="http://www.itfinity.net/soap/exrates/exrates.xsd"
    operation="GetRate"
    resulttype="double"
    result="myresult">
    <web:param name="fromCurr" value="<%="str1%"/>" />
    <web:param name="ToCurr" value="<%="str2%"/>" />
</web:invoke>

<!-- Display the web service result. -->
<%= pageContext.getAttribute("myresult") %>
```

Data source access

Another difference between Flex and other web application technologies is that you never communicate directly with a data source in Flex. You use a Flex service component to connect to a server-side service that interacts with the data source.

The following example shows one way to access a data source directly in a ColdFusion page:

```
<CFQUERY DATASOURCE="Dsn"
    NAME="myQuery">
    SELECT * FROM table
</CFQUERY>
```

To get similar functionality in Flex, use an `HTTPService`, a `WebService`, or a `RemoteObject` component to call a server-side object that returns results from a data source.

HTTP services and web services

You define an `HTTPService` or `WebService` component in MXML or ActionScript. After defining the component, use the component to call a destination defined on the server and process any results.

Using destinations

You typically connect an RPC component to a destination defined in the `services-config.xml` file or a file that it includes by reference, such as the `proxy-config.xml` file. A destination definition is a named service configuration that provides server-proxied access to an RPC service. A destination is the actual service or object that you want to call.

Destination definitions provide centralized administration of RPC services. They also enable you to use basic or custom authentication to secure access to destinations. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging capability to log RPC service traffic.

You have the option of omitting the destination and connecting to HTTP and web services directly by specifying the URL of the service. For more information, see “[Using `HTTPService` and `WebService` without a destination](#)” on page 145. However, you must define a destination when using the `RemoteObject` component.

You configure HTTP services and web services as `HTTPProxyService` destinations. The following example shows a `HTTPProxyService` destination definition for an HTTP service in the `proxy-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">
    ...
    <destination id="myHTTPService">
        <properties>
            <!-- The endpoint available to the http proxy service -->
            <url>http://www.mycompany.com/services/myservlet</url>
            <!-- Wildcard endpoints available to the http proxy services -->
            <dynamic-url>http://www.mycompany.com/services/*</dynamic-url>
        </properties>
    </destination>
</service>
```

Using an RPC component with a server-side destination

The `destination` property of an RPC component references a destination configured in the `proxy-config.xml` file. A destination specifies the RPC service class or URL, the transport channel to use, the adapter with which to access the RPC service, and security settings.

To declare a connection to a destination in MXML, set the `id` and `destination` properties in the RPC component. The `id` property is required for calling the services and handling service results. The following example shows `HTTPService` and `WebService` component declarations in MXML:

```
<?xml version="1.0"?>
<!!-- ds\rpc\RPCMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService
        id="yahoo_web_search"
        useProxy="true"
        destination="catalog"/>

    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"/>
</mx:Application>
```

When you use a destination with an `HTTPService` or `WebService` component, you set its `useProxy` property to `true` to configure it to use the `HTTPProxyService`. The `HTTPProxyService` and its adapters provide functionality that lets applications access HTTP services and web services on different domains. Additionally, the `HTTPProxyService` lets you limit access to specific URLs and URL patterns, and provide security.

Note: *Setting the `destination` property of the `HTTPService` or `WebService` component automatically sets the `useProxy` property to `true`.*

When you do not have LiveCycle Data Services or do not require the functionality provided by the `HTTPProxyService`, you can bypass it. You bypass the proxy by setting the `useProxy` property of an `HTTPService` or `WebService` component to `false`, which is the default value. Also set the `HTTPService.url` property to the URL of the HTTP service, or set the `WebService.wsdl` property to the URL of the WSDL document. The `RemoteObject` component does not define the `useProxy` property; you always use a LiveCycle Data Services destination with the `RemoteObject` component.

Configuring a destination

You configure a destination in a service definition in the `proxy-config.xml` file. The following example shows a basic server-side configuration for a `WebService` component in the `proxy-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">

    <properties>
        <connection-manager>
            <max-total-connections>100</max-total-connections>
            <default-max-connections-per-host>2</default-max-connections-per-host>
        </connection-manager>
        <allow-lax-ssl>true</allow-lax-ssl>
    </properties>

    <!-- Channels are typically defined in the services-config.xml file. -->
    <default-channels>
        <channel ref="my-http"/>
        <channel ref="my-amf"/>
    </default-channels>

    <!-- Define the adapters used by the different destinations. -->
    <adapters>
        <adapter-definition id="http-proxy"
            class="flex.messaging.services.http.HTTPProxyAdapter"
            default="true"/>
        <adapter-definition id="soap-proxy"
            class="flex.messaging.services.http.SOAPProxyAdapter"/>
    </adapters>

    <!-- HTTPService destination uses the default adapter. -->
    <destination id="catalog">
        <properties>
            <url>/<context.root>/testdrive-httpservice/catalog.jsp</url>
        </properties>
    </destination>

    <!-- WebService destination uses the SOAPAdapter. -->
    <destination id="ws-catalog">
        <properties>
            <wsdl>http://livecycledata.org/services/ProductWS?wsdl</wsdl>
            <soap>http://livecycledata.org/services/ProductWS/*</soap>
        </properties>
        <adapter ref="soap-proxy"/>
    </destination>
</service>
```

HTTPService and WebService components connect to HTTPProxyService destinations. Therefore, the `class` attribute of the `<service>` tag specifies the `HTTPProxyService` class.

The adapter is server-side code that interacts with the remote service. Specify the `HTTPProxyAdapter` with a destination defined for the `HTTPService` component, and the `SOAPAdapter` for the `WebService` component.

This destination uses an HTTP or an Action Message Format (AMF) message channel for transporting data. Optionally, it could use one of the other supported message channels. Message channels are defined in the `services-config.xml` file, in the `channels` section under the `services-config` element. For more information, see “[Channels and endpoints](#)” on page 38.

You use the `url` and `dynamic-url` elements to configure HTTP service URLs in a destination. These elements define which URLs are permitted for a destination. The following table describes those elements:

Element	Description
url	(Optional) Default URL. The HTTPService URL could be at a domain that is different from the one hosting your SWF file and therefore cannot be requested directly from Adobe Flash Player due to the sandbox security restrictions the player enforces. When that is the case, you can use the Proxy Service to proxy the request. You establish a set of URLs that you permit it to proxy for you. In a Proxy Service destination you configure the URL in the url property to instruct the Proxy Service to allow requests to be proxied to this URL. The url property also lets you avoid hard coding the WSDL URL into your MXML file by specifying the destination name instead of the WSDL URL.
dynamic-url	(Optional) HTTP service URL patterns. You can use more than one dynamic-url entry to specify multiple URL patterns. Flex matches these values against url property values that you specify in client-side service tags or ActionScript code.

You use the wsdl and soap elements to configure web service URLs in a destination. These elements define which URLs are permitted for a destination. The following table describes those elements:

Element	Description
wsdl	(Optional) Default WSDL URL. When you use a WebService component in a Flex application, the first thing that happens is an HTTP GET request is made to a URL to load the WSDL document for the web service. The URL could be at a domain that is different from the one hosting your SWF file and therefore cannot be requested directly from Adobe Flash Player due to the sandbox security restrictions the player enforces. When that is the case, you can use the Proxy Service to proxy the request. You establish a set of URLs that you permit it to proxy for you. In a Proxy Service destination you configure the WSDL URL in the wsdl property to instruct the Proxy Service to allow requests to be proxied to this URL. The wsdl property also lets you avoid hard coding the WSDL URL into your MXML file by specifying the destination name instead of the WSDL URL.
soap	SOAP endpoint URL patterns that would typically be defined for each operation in the WSDL document. In a WSDL document, there is a port section, usually at the end of the file, that describes the location of the SOAP endpoints to handle web service requests. These URLs are used in subsequent HTTP POST requests that send SOAP-formatted requests and corresponding SOAP-formatted responses. You add these URLs using the soap properties in the destination configuration. You typically use a soap property to define a SOAP endpoint URL pattern for each operation in a WSDL document. You use more than one soap entry to specify multiple SOAP endpoint patterns. You can use wildcards in these URLs because in more complex WSDLs you can have multiple services, multiple ports, and therefore multiple SOAP address locations. Example: <pre><soap>http://www.weather.gov/forecasts/xml/SOAP_server/*</soap></pre> Using a <soap>*</soap> element on an unprotected destination is not a good idea because you are effectively setting up a public anonymous relay. If you do want to use a very lenient wildcard, you can set a role-based J2EE security restriction and require authentication on the message broker so that only authenticated requests are processed. Note: If you use endpointURI property values in client-side service tags or ActionScript code, Flex matches the values specified in soap properties against those.

Using HTTPService and WebService with a default destination

In the following situation, a component uses a default destination:

- You set the useProxy property to true
- You do not set the destination property
- You set the HTTPService.url property or the WebService.wsdl property

The default destinations are named `DefaultHTTP` and `DefaultHTTPS`. The `DefaultHTTPS` destination is used when your `url` or `wsdl` property specifies an HTTPS URL.

By setting the `url` or `wsdl` property, you let the component specify the URL of the remote service, rather than specifying it in the destination. However, since the request uses the default destination, you can take advantage of LiveCycle Data Services. Therefore, you can use basic or custom authentication to secure access to the destination. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging capability to log remote service traffic.

Configure the default destinations in the `proxy-config.xml` file. Use one or more `dynamic-url` parameters to specify URL patterns for the HTTP service, or one or more `soap` parameters to specify URL patterns for the WSDL of the web service. The value of the `url` or `wsdl` property of the component must match the specified pattern.

The following example shows a default destination definition that specifies a `dynamic-url` value:

```
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">
  ...
  <destination id="DefaultHTTP">
    <channels>
      <channel ref="my-amf"/>
    </channels>

    <properties>
      <dynamic-url>http://mysite.com/myservices/*</dynamic-url>
    </properties>
    ...
  </destination>
</service>
```

Therefore, set the `HTTPService.url` property to a URL that begins with the pattern `http://mysite.com/myservices/`. Setting the `HTTPService.url` property to any other value causes a `fault` event.

The following table describes elements that you use to configure the default destinations:

Parameter	Description
<code>dynamic-url</code>	Specifies the URL pattern for the HTTP service. You can use one or more <code>dynamic-url</code> parameters to specify multiple patterns. Any request from the client must match one of the patterns.
<code>soap</code>	Specifies the URL pattern for a WSDL or the location of a web service. You can use one or more <code>soap</code> parameters to specify multiple patterns. The URL of the WSDL, or the URL of the web service, must match one of the patterns. If you specify a URL for the <code>wsdl</code> property, the URL must match a pattern,

Configuring the Proxy Service

Configure the Proxy Service by using the `proxy-config.xml` file. The `service` parameter contains a `properties` element that you use to configure the Apache connection manager, self-signed certificates for SSL, and external proxies. The following table describes elements that you use to configure the Proxy Service:

Element	Description
connection-manager	Contains the <code>max-total-connections</code> and <code>default-max-connections-per-host</code> elements. The <code>max-total-connections</code> element controls the maximum total number of concurrent connections that the proxy supports. If the value is greater than 0, LiveCycle Data Services uses a multithreaded connection manager for the underlying Apache HttpClient proxy. The <code>default-max-connections-per-host</code> element sets the default number of connections allowed for each host in an environment that uses hardware clustering.
content-chunked	Specifies whether to use chunked content. The default value is <code>false</code> . Flash Player does not support chunked content.
allow-lax-ssl	Set to <code>true</code> when using SSL to allow self-signed certificates. Do not set to <code>true</code> in a production environment.
external-proxy	Specifies the location of an external proxy, as well as a user name and a password, when the Proxy Service must contact an external proxy before getting access to the Internet. The properties of <code>external-proxy</code> depend on the external proxy and the underlying Apache HttpClient proxy. For more information on the Apache HttpClient, see the Apache website.

The following example shows a Proxy Service configuration:

```
<service id="proxy-service" class="flex.messaging.services.HTTPProxyService">

    <!-- Define channels and destinations. -->

    <properties>
        <connection-manager>
            <max-total-connections>100</max-total-connections>
            <default-max-connections-per-host>2
            </default-max-connections-per-host>
        </connection-manager>

        <!-- Allow self-signed certificates. Do not use in production -->
        <allow-lax-ssl>true</allow-lax-ssl>

        <!-- Connection settings for an external proxy. -->
        <external-proxy>
            <server>10.10.10.10</server>
            <port>3128</port>
            <nt-domain>mycompany</nt-domain>
            <username>flex</username>
            <password>flex</password>
        </external-proxy>
    </properties>
</service>
```

Using HTTPService and WebService without a destination

You can connect to HTTP services and web services without configuring a destination. To do so, you set the `HTTPService.url` property or the `WebService.wsdl` property instead of setting the `destination` property. Additionally, set the `useProxy` property of the component to `false` to bypass the `HTTPProxyService`. When the `useProxy` property is set to `false`, the component communicates directly with the service based on the `url` or `wsdl` property value.

When you set the `useProxy` property to `true` for the `HTTPService` component, you can use the HTTP HEAD, OPTIONS, TRACE, and DELETE methods. However, when you do not go through the `HTTPProxyService`, you can use only HTTP GET or POST methods. By default, the `HTTPService` method uses the GET method.

Note: If you bypass the proxy, and the status code of the HTTP response is not a success code from 200 through 299, Flash Player cannot access any data in the body of the response. For example, a server sends a response with an error code of 500 with the error details in the body of the response. Without a proxy, the body of the response is inaccessible by the Flex application.

Connecting to a service in this manner requires that at least one of the following is true:

- The service is in the same domain as your Flex application.
- A `crossdomain.xml` (cross-domain policy) file is installed on the web server hosting the RPC service that allows access from the domain of the application. For more information, see the Flex documentation.

The following examples show MXML tags for declaring `HTTPService` and `WebService` components that directly reference RPC services. The `id` property is required for calling the services and handling service results. In these examples, the `useProxy` property is not set in the tags. Therefore, the components use the default `useProxy` value of `false` and contact the services directly:

```
<?xml version="1.0"?>
<!- - ds\rpc\RPCNoServer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService
        id="yahoo_web_search"
        url="http://api.search.yahoo.com/WebSearchService/V1/webSearch"/>

    <mx:WebService
        id="macr_news"
        wsdl="http://ws.invesbot.com/companysearch.asmx?wsdl"/>
</mx:Application>
```

Defining and invoking an `HTTPService` component

Use the `HTTPService` components in your client-side application to make an HTTP request to a URL. The following examples show `HTTPService` component declarations in MXML and in ActionScript. Regardless of how you define the component, you send a request to the destination by calling the `HTTPService.send()` method. To handle the results, you use the `result` and `fault` event handlers:

```
<?xml version="1.0"?>
<!- ds\rpc\HttpService.mxml. -->
&ltmx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="useHttpService();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.http.mxml.HTTPService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            private var asService:HTTPService
            // Define the HTTPService component in ActionScript.
            public function useHttpService():void {
                asService = new HTTPService();
                asService.method = "POST";
                asService.useProxy = true;
                asService.destination = "catalog";
                asService.addEventListener("result", httpResult);
                asService.addEventListener("fault", httpFault);
            }

            public function httpResult(event:ResultEvent):void {
                //Do something with the result.
            }

            public function httpFault(event:FaultEvent):void {
                var faultstring:String = event.fault.faultString;
                Alert.show(faultstring);
            }
        ]]>
    </mx:Script>

    <!- Define the HTTPService component in MXML. -->
    &ltmx:HTTPService
        id="mxmlService"
        method="POST"
        useProxy="true"
        destination="catalog"
        result="httpResult(event);"
        fault="httpFault(event);"/>

    &ltmx:Button label="MXML" click="mxmlService.send();"/>
    &ltmx:Button label="AS" click="asService.send();"/>

</mx:Application>
```

Defining and invoking a WebService component

Use the WebService components in your client-side application to make a SOAP request. The following example defines a WebService component in MXML and ActionScript:

```
<?xml version="1.0"?>
<!- ds\rpc\WebServiceExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="useWebService();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.soap.mxml.WebService;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            private var asService:WebService;

            // Define the WebService component in ActionScript.
            public function useWebService():void {
                asService = new WebService();
                asService.destination = "ws-catalog";
                asService.addEventListener("result", wsResult);
                asService.addEventListener("fault", wsFault);
                asService.loadWSDL();
            }

            public function wsResult(event:ResultEvent):void {
                //Do something with the result.
            }

            public function wsFault(event:FaultEvent):void {
                var faultstring:String = event.fault.faultString;
                Alert.show(faultstring);
            }
        ]]>
    </mx:Script>

    <!- Define the WebService component in MXML. -->
    <mx:WebService
        id="mxmlService"
        destination="ws-catalog"
        result="wsResult(event);"
        fault="wsFault(event);"/>

    <mx:Button label="MXML" click="mxmlService.getProducts();"/>
    <mx:Button label="AS" click="asService.getProducts();"/>

</mx:Application>
```

The destination specifies the WSDL associated with this web service. A web service can expose multiple methods, corresponding to multiple operations. In this example, you directly call the `getProducts()` operation of the web service in response to a `click` event of a `Button` control.

Important: Notice that the ActionScript version of the `WebService` component calls the `WebService.loadWSDL()` method to load the WSDL. This is required when you create the `WebService` component in ActionScript. This method is called automatically when you define the component in MXML.

Using an Operation object with the WebService component

Because a single WebService component can invoke multiple operations on the web service, the component requires a way to represent information specific to each operation. Therefore, for every operation, the component creates an mx.rpc.soap.mxxml.Operation object.

The name of the Operation object corresponds to the name of the operation. From the example shown in “[Defining and invoking a WebService component](#)” on page 147, access the Operation object that corresponds to the `getProducts()` operation by accessing the Operation object named `getProducts`, as the following code shows:

```
// The Operation object has the same name as the operation, without the trailing parentheses.  
var myOP:Operation = mxxmlService.getProducts;
```

The Operation object contains properties that you use to set characteristics of the operation, such as the arguments passed to the operation, and to hold any data returned by the operation. You access the returned data by using the `Operation.lastResult` property.

Invoke the operation by referencing it relative to the WebService component, as the following example shows:

```
mxxmlService.getProducts();
```

Alternatively, invoke an operation by calling the `Operation.send()` method, as the following example shows:

```
mxxmlService.getProducts.send();
```

Defining multiple operations for the WebService component

When a web service defines multiple operations, you define multiple operations for the WebService component and specify the attributes for each operation, as the following example shows:

```
<mx:WebService  
    id="mxxmlService"  
    destination="ws-catalog"  
    result="wsResult(event);">  
    <mx:operation name="getProducts" fault="getPFault(event);"/>  
    <mx:operation name="updateProdcut" fault="updatePFault(event);"/>  
    <mx:operation name="deleteProduct" fault="deletePFault(event);"/>  
</mx:WebService>
```

The `name` property of an `<mx:operation>` tag must match one of the web service operation names. The WebService component creates a separate Operation object for each operation.

Each operation can rely on the event handlers and characteristics defined by the WebService component. However, the advantage of defining the operations separately is that each operation can specify its own event handlers, its own input parameters, and other characteristics. In this example, the WebService component defines the result handler for all three operations, and each operation defines its own fault handler. For more information, see “[Handling service events](#)” on page 171 and “[Passing parameters to a service](#)” on page 149.

Passing parameters to a service

Flex provides two ways to pass parameters to a service call: *explicit parameter passing* and *parameter binding*. With explicit parameter passing, pass properties in the method that calls the service. With parameter binding, use data binding to populate the parameters.

Using explicit parameter passing

When you use explicit parameter passing, you provide input to a service in the form of parameters to an ActionScript function. This way of calling a service closely resembles the way that you call methods in Java.

Explicit parameter passing with HTTPService components

When you use explicit parameter passing with an HTTPService component, you specify an object that contains name-value pairs as an argument to the `send()` method. A `send()` method parameter must be a simple base type such as `Object`. You cannot use complex nested objects because there is no generic way to convert them to name-value pairs.

The following examples show two ways to call an HTTP service using the `send()` method with a parameter.

```
<?xml version="1.0"?>
<!!-- ds\rpc\RPCSend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function callService():void {
                var params:Object = new Object();
                params.param1 = 'val1';
                myService.send(params);
            }
        ]]>
    </mx:Script>

    <mx:HTTPService
        id="myService"
        destination="catalog"
        useProxy="true"/>

    <!-- HTTP service call with a send() method that takes
        a variable as its parameter. The value of the variable is an Object. -->
    <mx:Button label="send() with variable" click="myService.send({param1: 'val1'});" />

    <!-- HTTP service call with an object as a send() method parameter
        that provides query parameters. -->
    <mx:Button label="send() with query params" click="callService();"/>
</mx:Application>
```

Explicit parameter passing with WebService components

When using the WebService component, you call a service by directly calling the `service` method, or by calling the `Operation.send()` method of the `Operation` object that represents the operation. When you use explicit parameter passing, you specify the parameters as arguments to the method that you use to invoke the operation.

The following example shows MXML code for declaring a WebService component and calling a service using explicit parameter passing in the `click` event listener of a Button control. A ComboBox control provides data to the service, and event listeners handle the service-level result and fault events:

```
<?xml version="1.0"?>
<!- ds\rpc\RPCParamPassingWS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:WebService
        id="employeeWS"
        destination="SalaryManager"/>

    <mx:ComboBox id="dept" width="150">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:source>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:source>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:ComboBox>

    <mx:Button label="Get Employee List"
        click="employeeWS.getList(dept.selectedItem.data);"/>
</mx:Application>
```

Using data binding to pass parameters

Parameter binding lets you copy data from user interface controls or models to request parameters. You typically declare data bindings in MXML. However, you can also define them in ActionScript. For more information about data binding, see the Flex documentation.

Binding with HTTPService components

Parameters to an HTTPService component correspond to query parameters of the requested URL. When an HTTP service takes query parameters, you can specify them by using the `request` property. The `request` property takes an Object of name-value pairs used as parameters to the URL. The names of the properties must match the names of the query parameters that the service expects. If the `HTTPService.contentType` property is set to `application/xml`, the `request` property must be an XML document.

When you use parameter binding, you call a service by using the `send()` method but specify no arguments to the method. The `HTTPService` component automatically adds the parameters specified by the `request` property to the request.

Note: If you do not specify a parameter to the `send()` method, the `HTTPService` component uses any query parameters specified in an `<mx:request>` tag.

The following example binds the selected data of a `ComboBox` control to the `request` property:

```
<?xml version="1.0"?>
<!!-- ds\rpc\HttpServiceParamBind.mxml. Compiles -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HTTPService
        id="employeeSrv"
        destination="catalog">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </mx:HTTPService>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeSrv.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{employeeSrv.lastResult.employees.employee}"
        width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Binding with WebService components

When you use parameter binding with a WebService component, you typically declare an operation by using the `operation` property. Each `operation` property corresponds to an instance of the `Operation` class, which defines a `request` property that contains the XML nodes that the operation expects.

The following example binds the data of a selected ComboBox item to the `getList()` operation. When you use parameter binding, you call a service by using the `send()` method with no arguments:

```
<?xml version="1.0"?>
<! -- ds\rpc\WebServiceParamBind.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.utils.ArrayUtil;
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:ArrayCollection
        id="employeeAC"
        source="{ArrayUtil.toArray(employeeWS.getList.lastResult)}"/>

    <mx:WebService
        id="employeeWS"
        destination="wsDest"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString);">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeWS.getList.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="to_email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

You can manually specify an entire SOAP request body in XML with all of the correct namespace information defined in the `request` property. Set the value of the `format` attribute of the `request` property to `xml`, as the following example shows:

```
<?xml version="1.0"?>
<!!-- ds\rpc\WebServiceSOAPRequest.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">
    <mx:WebService id="ws" wsdl="http://api.google.com/GoogleSearch.wsdl"
        useProxy="true">
        <mx:operation name="doGoogleSearch" resultFormat="xml">
            <mx:request format="xml">
                <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                    <key xsi:type="xsd:string">XYZ123</key>
                    <q xsi:type="xsd:string">Balloons</q>
                    <start xsi:type="xsd:int">0</start>
                    <maxResults xsi:type="xsd:int">10</maxResults>
                    <filter xsi:type="xsd:boolean">true</filter>
                    <restrict xsi:type="xsd:string"/>
                    <safeSearch xsi:type="xsd:boolean">false</safeSearch>
                    <lr xsi:type="xsd:string" />
                    <ie xsi:type="xsd:string">latin1</ie>
                    <oe xsi:type="xsd:string">latin1</oe>
                </ns1:doGoogleSearch>
            </mx:request>
        </mx:operation>
    </mx:WebService>
</mx:Application>
```

Using capabilities specific to WebService components

Flex applications can interact with web services that define their interfaces in a Web Services Description Language 1.1 (WSDL 1.1) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages. The Flex web service API generally supports SOAP 1.1, XML Schema 1.0 (versions 1999, 2000 and 2001), and WSDL 1.1 rpc-encoded, and rpc-literal, document-literal (bare and wrapped style parameters). The two most common types of web services use RPC-encoded or document-literal SOAP bindings; the terms *encoded* and *literal* indicate the type of WSDL-to-SOAP mapping that a service uses.

Note: Flex does not support the following XML schema types: *union*, *default*, or *list*. Flex also does not support the following data types: *duration*, *gMonth*, *gYear*, *gYearMonth*, *gDay*, *gMonthDay*, *Name*, *Qname*, *NCName*, *anyURI*, or *language*. These data types are treated as *Strings* and not validated. Flex supports any URL but treats it like a String.

Flex applications support web service requests and results that are formatted as Simple Object Access Protocol (SOAP) messages. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

Adobe Flash Player operates within a security sandbox that limits what Flex applications and other Flash Player applications can access over HTTP. Flash Player applications are only allowed HTTP access to resources on the same domain and by the same protocol from which they were served. This restriction presents a problem for web services, because they are typically accessed from remote locations. The Proxy Service, available in LiveCycle Data Services, intercepts requests to remote web services, redirects the requests, and then returns the responses to the client.

If you are not using LiveCycle Data Services, you can access web services in the same domain as your Flex application. Or, a crossdomain.xml file that allows access from the domain of the application must be installed on the web server hosting the RPC service. For more information, see the Adobe Flex 3 documentation.

Note: If you are using Flash Player version 9.0.124.0 or later, the `crossdomain.xml` file has a new tag called `<allow-http-request-headers-from>` that you use to set header-sending rights. For web services, make sure to set the `headers` attribute of the `<allow-http-request-headers-from>` tag to `SOAPAction`. For more information, see http://www.adobe.com/devnet/flashplayer/articles/flash_player9_security_update.html and <http://kb.adobe.com/selfservice/viewContent.do?externalId=kb403185&sliceId=2>.

Reading WSDL documents

View a WSDL document in a web browser, a simple text editor, an XML editor, or a development environment such as Adobe Dreamweaver, which contains a built-in utility for displaying WSDL documents in an easy-to-read format.

For a complete description of the format of a WSDL document, see <http://www.w3.org/TR/wsdl>.

RPC-oriented operations and document-oriented operations

A WSDL file can specify either remote procedure call-oriented (RPC) or document-oriented (document/literal) operations. Flex supports both operation styles.

When calling an RPC-oriented operation, a Flex application sends a SOAP message that specifies an operation and its parameters. When calling a document-oriented operation, a Flex application sends a SOAP message that contains an XML document.

In a WSDL document, each `<port>` tag has a `binding` property that specifies the name of a particular `<soap:binding>` tag, as the following example shows:

```
<binding name="InstantMessageAlertSoap" type="s0:InstantMessageAlertSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document"/>
```

The `style` property of the associated `<soap:binding>` tag determines the operation style. In this example, the `style` is `document`.

Any operation in a service can specify the same style or override the style that is specified for the port associated with the service, as the following example shows:

```
<operation name="SendMSN">
    <soap:operation soapAction="http://www.bindingpoint.com/ws/imalert/
        SendMSN" style="document"/>
```

Stateful web services

LiveCycle Data Services can maintain the state of web service endpoints. If the web service uses cookies to store session information, LiveCycle Data Services uses Java server sessions to maintain the state of the web service. This capability acts as an intermediary between Flex applications and web services. It adds the identity of an endpoint to whatever the endpoint passes to a Flex application. If the endpoint sends session information, the Flex application receives it. This capability requires no configuration; it is not supported for destinations that use the RTMP channel when using the `HTTPProxyService`.

Working with SOAP headers

A SOAP header is an optional tag in a SOAP envelope that usually contains application-specific information, such as authentication information.

Adding SOAP headers to web service requests

Some web services require that you pass a SOAP header when you call an operation. Add a SOAP header to all web service operations or individual operations by calling the `addHeader()` or `addSimpleHeader()` method of the `WebService` or `Operation` object.

When you use the `addHeader()` method, you first create `SOAPHeader` and `QName` objects separately. The `addHeader()` method has the following signature:

```
addHeader(header:mx.rpc.soap.SOAPHeader):void
```

To create a `SOAPHeader` object, you use the following constructor:

```
SOAPHeader(qname:QName, content:Object)
```

The `content` parameter of the `SOAPHeader()` constructor is a set of name-value pairs based on the following format:

```
{name1:value1, name2:value2}
```

To create the `QName` object in the first parameter of the `SOAPHeader()` method, you use the following constructor:

```
QName(uri:String, localName:String)
```

The `addSimpleHeader()` method is a shortcut for a single name-value SOAP header. When you use the `addSimpleHeader()` method, you create `SOAPHeader` and `QName` objects in parameters of the method. The `addSimpleHeader()` method has the following signature:

```
addSimpleHeader(qnameLocal:String, qnameNamespace:String, headerName:String,  
headerValue:Object):void
```

The `addSimpleHeader()` method takes the following parameters:

- `qnameLocal` is the local name for the header `QName`.
- `qnameNamespace` is the namespace for the header `QName`.
- `headerName` is the name of the header.
- `headerValue` is the value of the header. This value can be a String if it is a simple value, an Object that undergoes basic XML encoding, or XML if you want to specify the header XML yourself.

The following calls to the `addSimpleHeader()` and `addHeader()` methods are equivalent:

```
addHeader(new QName(qNs,qLocal), {name:val});  
addSimpleHeader(qLocal, qNs, name, val);
```

Both methods add a `SOAPHeader` object to a collection of headers. Each `SOAPHeader` is encoded as follows:

```
<qnamePrefix:qnameLocal>  
    content  
</qnamePrefix:qnameLocal>
```

If the `content` parameter contains simple data, its String representation is used. If it contains an Object, its structure is converted to XML. For example, if the `content` parameter passed to the method contains the following data:

```
{name:value}
```

The `SOAPHeader` is encoded as follows:

```
<qnamePrefix:qnameLocal>  
    <headerName>headerValue</headerName>  
</qnamePrefix:qnameLocal>
```

If the `content` parameter contains a property with the same name as `qnameLocal`, the value of that property is used as the header content. Therefore, if `qnameLocal` equals `headerName`, the `SOAPHeader` object is encoded as follows:

```
<qnamePrefix:headerName>
    headerValue
</qnamePrefix:headerName>
```

The code in the following example shows how to use the `addHeader()` method and the `addSimpleHeader()` method to add a SOAP header. The methods are called in the `headers()` function, and the event listener is assigned in the `load` property of an WebService component:

```
<?xml version="1.0"?>
<!!-- ds\rpc\WebServiceAddHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.SOAPHeader;

            private var header1:SOAPHeader;
            private var header2:SOAPHeader;
            public function headers():void {

                // Create QName and SOAPHeader objects.
                var q1:QName = new QName("http://soapinterop.org/xsd", "Header1");
                header1 = new SOAPHeader(q1, {string:"bologna",int:"123"});
                header2 = new SOAPHeader(q1, {string:"salami",int:"321"});

                // Add the header1 SOAP Header to all web service requests.
                ws.addHeader(header1);

                // Add the header2 SOAP Header to the getSomething operation.
                ws.getSomething.addHeader(header2);

                // Within the addSimpleHeader method,
                // which adds a SOAP header to web
                // service requests, create SOAPHeader and QName objects.
                ws.addSimpleHeader("header3", "http://soapinterop.org/xsd", "foo", "bar");
            }
        ]]>
    </mx:Script>

    <mx:WebService id="ws"
        destination="wsDest"
        load="headers();"/>
</mx:Application>
```

Clearing SOAP headers

Use the `clearHeaders()` method of a `WebService` or `Operation` object to remove SOAP headers that you added to the object, as the following example shows:

```
<?xml version="1.0"?>
<!!-- ds\rpc\WebServiceClearHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- The value of the destination property is for demonstration only
        and is not a real destination. -->
    <mx:Script>
        <! [CDATA[
            import mx.rpc.*;
            import mx.rpc.soap.SOAPHeader;

            private function headers():void {
                // Create QName and SOAPHeader objects.
                var q1:QName = new QName("Header1", "http://soapinterop.org/xsd");
                var header1:SOAPHeader = new SOAPHeader(q1, {string:"bologna",int:"123"});
                var header2:SOAPHeader = new SOAPHeader(q1, {string:"salami",int:"321"});
                // Add the header1 SOAP Header to all web service request.
                ws.addHeader(header1);
                // Add the header2 SOAP Header to the getSomething operation.
                ws.getSomething.addHeader(header2);

                // Within the addSimpleHeader method, which adds a SOAP header to all
                // web service requests, create SOAPHeader and QName objects.
                ws.addSimpleHeader("header3","http://soapinterop.org/xsd", "foo", "bar");
            }

            // Clear SOAP headers added at the WebService and Operation levels.
            private function clear():void {
                ws.clearHeaders();
                ws.getSomething.clearHeaders();
            }
        ]]>
    </mx:Script>

    <mx:WebService id="ws"
        destination="wsDest"
        load="headers();"/>

    <mx:HBox>
        <mx:Button label="Clear headers and run again"
            click="clear();"/>
    </mx:HBox>
</mx:Application>
```

Redirecting a web service to a different URL

Some web services require that you change to a different endpoint URL after you process the WSDL and make an initial call to the web service. For example, suppose you want to use a web service that requires you to pass security credentials. After you call the web service to send login credentials, it accepts the credentials and returns the actual endpoint URL that is required to use the business operations. Before calling the business operations, change the `endpointURI` property of your `WebService` component.

The following example shows a `result` event listener that stores the endpoint URL that a web service returns in a variable, and then sets the endpoint URL for subsequent requests:

```
public function onLoginResult(event:ResultEvent):void {  
  
    //Extract the new service endpoint from the login result.  
    var newServiceURL = event.result.serverUrl;  
  
    // Redirect all service operations to the URL received in the login result.  
    serviceName.endpointURI=newServiceURL;  
}
```

A web service that requires you to pass security credentials can also return an identifier that you must attach in a SOAP header for subsequent requests; for more information, see “[Working with SOAP headers](#)” on page 155.

Remote objects

You declare RemoteObject components in MXML or ActionScript to connect to remote services.

Note: This documentation describes how to connect to Java classes in conjunction with Livecycle Data Services. For information about connecting to PHP or ColdFusion, see the Flex and ColdFusion documentation.

A destination for a RemoteObject component is a Java class defined as the source of a Remoting Service destination. Destination definitions provide centralized administration of remote services. They also enable you to use basic or custom authentication to secure access to destinations. You can choose from several different transport channels, including secure channels, for sending data to and from destinations. Additionally, you can use the server-side logging capability to log remote service traffic.

You can also use RemoteObject components with PHP and .NET objects in conjunction with third-party AMF implementations.

Remoting Service channels

With the Remoting Service, you often use an AMFChannel. The AMFChannel uses binary AMF encoding over HTTP. If binary data is not allowed, then you can use an HTTPChannel, which is AMFX (AMF in XML) over HTTP. Message channels are typically defined in the services-config.xml file, in the channels section under the services-config element. For more information on channels, see “[Client and server architecture](#)” on page 27.

Using a RemoteObject component

The following example shows a RemoteObject component that connects to a destination, sends a request to the data source in the click event of a Button control, and displays the result data in the text property of a TextArea control:

```
<?xml version="1.0"?>
<!- ds\rpc\RPCIntroExample1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the TextArea control.
                outputResult.text=remoteService.getData.lastResult.prop1;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!- Connect to a service destination.-->
    <mx:RemoteObject id="remoteService"
        destination="census"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!- Provide input data for calling the service. -->
    <mx:TextInput id="inputText"/>

    <!- Call the web service, use the text in a TextInput control as input data.-->
    <mx:Button click="remoteService.getData(inputText.text)"/>

    <!- Display results data in the user interface. -->
    <mx:TextArea id="outputResult"/>
</mx:Application>
```

Defining remote Java objects

One difference between Remoting Service destinations and HTTP service and web service destinations is that in Remoting Service destinations you host the remote Java object in your LiveCycle Data Services web application and reference it by using a destination. With HTTP service and web service destinations, you typically configure the destination to access a remote service, external to the web application. A developer is responsible for writing and compiling the Java class and adding it to the web application classpath by placing it in the WEB-INF\classes or WEB-INF\lib directory.

You can use any plain old Java object (POJO) that is available in the web application classpath as the source of the Remoting Service destination. The class must have a zero-argument constructor so that LiveCycle Data Services can construct an instance.

The following example shows a Remoting Service destination definition in the remoting-config.xml file. The `source` element specifies the fully qualified name of a class in the classpath of the web application.

```
<destination id="census">
  <properties>
    <source>flex.samples.census.CensusService</source>
  </properties>
</destination>
```

The following example shows the corresponding source code of the Java class that is referenced in the destination definition:

```
package flex.samples.census;

import java.util.ArrayList;
import java.util.List;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import flex.samples.ConnectionHelper;

public class CensusService
{
    public List getElements(int begin, int count)
    {

        long startTime = System.currentTimeMillis();

        Connection c = null;
        List list = new ArrayList();

        String sql = "SELECT id, age, classofworker, education, maritalstatus, race,
                     sex FROM census WHERE id > ? AND id <= ? ORDER BY id ";

        try {

            c = ConnectionHelper.getConnection();
            PreparedStatement stmt = c.prepareStatement(sql);
            stmt.setInt(1, begin);
            stmt.setInt(2, begin + count);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                CensusEntryVO ce = new CensusEntryVO();
                ce.setId(rs.getInt("id"));


```

```
        ce.setAge(rs.getInt("age"));
        ce.setClassOfWorker(rs.getString("classofworker"));
        ce.setEducation(rs.getString("education"));
        ce.setMaritalStatus(rs.getString("maritalstatus"));
        ce.setRace(rs.getString("race"));
        ce.setSex(rs.getString("sex"));
        list.add(ce);
    }
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        c.close();
    } catch (Exception ignored) {
    }
}
}

return list;
}
}
```

Placing Java objects in the classpath

The Remoting Service lets you access stateless and stateful objects that are in the classpath of the LiveCycle Data Services web application. Place class files in the WEB-INF\classes directory to add them to the classpath. Place Java Archive (JAR) files in the WEB-INF\lib directory to add them to the classpath.

Specify the fully qualified class name in the source property of a Remoting Service destination in the remoting-config.xml file. The class also must define a constructor that takes no arguments.

Converting ActionScript data to and from Java data

When you send data from a Flex application to a Java object, the data is automatically converted from an ActionScript data type to a Java data type. An object returned from a Java method is converted from Java to ActionScript. For a complete description of how data is converted, see “[Data serialization](#)” on page 79.

Reserved method names for the RemoteObject component

If a remote method has the same name as a method defined by the RemoteObject class, or by any of its parent classes, then you cannot call the remote method directly. The RemoteObject class defines the following method names; do not use these names as method names in your Java class:

```
disconnect()
getOperation()
hasOwnProperty()
initialized()
isPrototypeOf()
logout()
propertyIsEnumerable()
setCredentials()
setPropertyIsEnumerable()
setRemoteCredentials()
toString()
valueOf()
```

Do not begin Java method names with the underscore (_) character. If a remote method name matches a reserved method name, you can use the following ActionScript method with a RemoteObject or WebService component to return an Operation object that represents the method:

```
public function getOperation(name:String) :Operation
```

For example, if a remote method is called `hasOwnProperty()`, create an Operation object, as the following example shows:

```
public var myRemoteObject:RemoteObject = new RemoteObject();
myRemoteObject.destination = "ro-catalog";
public var op:Operation = myRemoteObject.getOperation("hasOwnProperty");
```

Invoke the remote method by using the `operation.send()` method, as the following example shows:

```
op.send();
```

RemoteObject endpoint property

The `RemoteObject endpoint` property lets you quickly specify an endpoint for a `RemoteObject` destination without referring to a services configuration file at compile time or programmatically creating a `ChannelSet`. It also overrides an existing `ChannelSet` if one is set for the `RemoteObject` service.

If the endpoint URL starts with `https`, a `SecureAMFChannel` is used. Otherwise, an `AMFChannel` is used. You can use two special tokens, `{server.name}` and `{server.port}`, in the endpoint URL to specify that the channel should use the server name and port that was used to load the SWF.

Note: The `endpoint` property is required for AIR applications that use the `RemoteObject` component.

Configuring a destination

You configure Remoting Service destinations in the Remoting Service definition in the `remoting-config.xml` file. The following example shows a basic server-side configuration for a Remoting Service in the `remoting-config.xml` file:

```
<service id="remoting-service"
    class="flex.messaging.services.RemotingService">

    <adapters>
        <adapter-definition id="java-object"
            class="flex.messaging.services.remoting.adapters.JavaAdapter"
            default="true"/>
    </adapters>

    <default-channels>
        <channel ref="samples-amf"/>
    </default-channels>

    <destination id="restaurant">
        <properties>
            <source>samples.restaurant.RestaurantService</source>
            <scope>application</scope>
        </properties>
    </destination>
</service>
```

The `class` attribute of the `<service>` tag specifies the `RemotingService` class. `RemoteObject` components connect to `RemotingService` destinations.

The adapter is server-side code that interacts with the Java class. Because you set the JavaAdapter as the default adapter, all destinations use it unless the destination explicitly specifies another adapter.

Use the `source` and `scope` elements of a Remoting Service destination definition to specify the Java object that the destination uses. Additionally, specify whether the destination is available in the `request` scope (stateless), the `application` scope, or the `session` scope. The following table describes these properties:

Element	Description
<code>source</code>	Fully qualified class name of the Java object (remote object).
<code>scope</code>	Indicates whether the object is available in the <code>request</code> scope, the <code>application</code> scope, or the <code>session</code> scope. Use the <code>request</code> scope when you configure a Remoting Service destination to access stateless objects. With the <code>request</code> scope, the server creates an instance of the Java class on each request. Use the <code>request</code> scope if you are storing the object in the application or session scope causes memory problems. When you use the <code>session</code> scope, the server creates an instance of the Java object once on the server for the session. For example, multiple tabs in the same web browser share the same session. If you open a Flex application in one tab, any copy of that application running in another tab accesses the same Java object. When you use the <code>application</code> scope, the server creates an instance of the Java object once on the server for the entire application. The default value is <code>request</code> .

For Remoting Service destinations, you can declare destinations that only allow invocation of methods that are explicitly included in an include list. Any attempt to invoke a method that is not in the `include-methods` list results in a fault. For even finer grained security, you can assign a security constraint to one or more of the methods in the `include-methods` list. If a destination-level security constraint is defined, it is tested first. Following that, the method-level constraints are checked. For more information, see “[Configuring a destination to use a security constraint](#)” on page 434.

Calling a service

Define the `RemoteObject` components in your client-side Flex application in MXML or ActionScript. The following example defines a `RemoteObject` component using both techniques:

```
<?xml version="1.0"?>
<!-- ds\rpc\ROInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="useRemoteObject();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.rpc.remoting.mxml.RemoteObject;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;

            public var asService:RemoteObject;

            public function useRemoteObject():void {
                asService = new RemoteObject();
                asService.destination = "ro-catalog";
                asService.getList.addEventListener("result", getListResultHandler);
                asService.addEventListener("fault", faultHandler);
                asService.getList();
            }
        ]]>
    </mx:Script>

```

```
}

public function getListResultHandler(event:ResultEvent):void {
    // Handle the result by accessing the event.result property.
}

public function faultHandler (event:FaultEvent):void {
    // Deal with event.fault.faultString, etc.
    Alert.show(event.fault.faultString, 'Error');
}

]]>
</mx:Script>

<!-- Define the RemoteObject component in MXML. -->
<mx:RemoteObject
    id="mxmlService"
    destination="ro-catalog"
    result="getListResultHandler(event);"
    fault="faultHandler(event);"/>

<mx:Button label="MXML" click="mxmlService.getList();"/>
<mx:Button label="AS" click="asService.getList();"/>
</mx:Application>
```

The destination specifies the Java class associated with the remote service. A Java class can expose multiple methods, corresponding to multiple operations. In this example, you directly call the `getList()` operation in response to a `click` event of a Button control.

Using the Operation class with the RemoteObject component

Because a single `RemoteObject` component can invoke multiple operations, the component requires a way to represent information specific to each operation. Therefore, for every operation, the component creates an `mx.rpc.remoting.mxml.Operation` object.

The name of the `Operation` object corresponds to the name of the operation. From the example shown above, you access the `Operation` object that corresponds to the `getList()` operation by accessing the `Operation` object named `getList`, as the following code shows:

```
// The Operation object has the same name as the operation, without the trailing parentheses.
var myOP:Operation = mxmlService.getList;
```

The `Operation` object contains properties that you use to set characteristics of the operation, such as the arguments passed to the operation, and to hold any data returned by the operation. You access the returned data by using the `Operation.lastResult` property.

Invoke the operation by referencing it relative to the `RemoteObject` component, as the following example shows:

```
mxmlService.getList();
```

Alternatively, invoke an operation by calling the `Operation.send()` method, as the following example shows:

```
mxmlService.getList.send();
```

Defining multiple operations for the RemoteObject component

When a service defines multiple operations, you define multiple methods for the `RemoteObject` component and specify the attributes for each method, as the following example shows:

```
<mx:RemoteObject
    id="mxmlService"
    destination="ro-catalog"
    result="roResult(event);">
    <mx:method name="getList" fault="getLFault(event);"/>
    <mx:method name="updateList" fault="updateLFault(event);"/>
    <mx:method name="deleteListItem" fault="deleteLIFault(event);"/>
</mx:RemoteObject>
```

The `name` property of an `<mx:method>` tag must match one of the operation names. The `RemoteObject` component creates a separate `Operation` object for each operation.

Each operation can rely on the event handlers and characteristics defined by the `RemoteObject` component. However, the advantage of defining the methods separately is that each operation can specify its own event handlers, its own input parameters, and other characteristics. In this example, the `RemoteObject` component defines the result handler for all three operations, and each operation defines its own fault handler. For an example that defines input parameters, see “[Using parameter binding to pass parameters to the `RemoteObject` component](#)” on page 167.

Note: *The Flex compiler defines the `method` property of the `RemoteObject` class; it does not correspond to an actual property of the `RemoteObject` class.*

Setting the concurrency property

The `concurrency` property of the `<mx:method>` tag indicates how to handle multiple calls to the same method. By default, making a new request to an operation or method that is already executing does not impact the existing request.

The following values of the `concurrency` property are permitted:

multiple Existing requests are not impacted and the developer is responsible for ensuring the consistency of returned data by carefully managing the event stream. The default value is `multiple`.

single Making only one request at a time is allowed on the method; additional requests made while a request is outstanding are immediately faulted on the client and are not sent to the server.

last Making a request causes the client to ignore a result or fault for any current outstanding request. Only the result or fault for the most recent request is dispatched on the client. This can simplify event handling in the client application, but be careful to only use this mode when you can safely ignore results or faults for requests.

Note: *The request referred to here is not the HTTP request. It is the method invocation request. If the transport between the client and server is HTTP, invocation requests are sent to the server within the body of HTTP requests and are processed on by the server. The last invocation request is not necessarily the last to be received by the server because in rare cases separate HTTP requests can travel over different routes through the network to the server, arriving in a different order than they were issued by the client.*

Passing parameters

Flex provides two ways to pass parameters to a service call: *explicit parameter passing* and *parameter binding*. With explicit parameter passing, pass properties in the method that calls the service. With parameter binding, use data binding to populate the parameters from user interface controls or models.

Explicit parameter passing with the `RemoteObject` component

The following example shows MXML code for declaring a `RemoteObject` component and calling a service using explicit parameter passing in the `click` event listener of a `Button` control. A `ComboBox` control provides data to the service.

```
<?xml version="1.0"?>
<!- ds\rpc\RPCParamPassing.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            [Bindable]
            public var empList:Object;
        ]]>
    </mx:Script>

    <mx:RemoteObject
        id="employeeRO"
        destination="SalaryManager"
        result="empList=event.result;"
        fault="Alert.show(event.fault.faultString, 'Error');"/>

    <mx:ComboBox id="dept" width="150">
        <mx:dataProvider>
            <mx:ArrayCollection>
                <mx:source>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:source>
            </mx:ArrayCollection>
        </mx:dataProvider>
    </mx:ComboBox>

    <mx:Button label="Get Employee List"
        click="employeeRO.getList(dept.selectedItem.data);"/>
</mx:Application>
```

Using parameter binding to pass parameters to the RemoteObject component

Parameter binding lets you copy data from user interface controls or models to request parameters. When you use parameter binding with RemoteObject components, you always declare operations in a RemoteObject component's `<mx:method>` tag. You then declare `<mx:arguments>` tags under an `<mx:method>` tag.

The order of the `<mx:arguments>` tags must match the order of the method parameters of the service. You can name argument tags to match the actual names of the corresponding method parameters as closely as possible, but it is not necessary.

Note: Defining multiple argument tags with the same name in an `<mx:arguments>` tag creates the argument as an Array with the specified name. The service call fails if the remote method is not expecting an Array as the only input parameter. No warning about this situation occurs when the application is compiled.

The following example uses parameter binding in a RemoteObject component's `<mx:method>` tag to bind the data of a selected ComboBox item to the `employeeRO.getList` operation when the user clicks a Button control. When you use parameter binding, you call a service by using the `send()` method with no parameters.

```
<?xml version="1.0"?>
<!- ds\rpc\ROParamBind2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>

    <mx:ArrayCollection id="employeeAC"
        source="{ArrayUtil.toArray(employeeRO.getList.lastResult)}"/>

    <mx:RemoteObject
        id="employeeRO"
        destination="roDest"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:method name="getList">
            <mx:arguments>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:arguments>
        </mx:method>
    </mx:RemoteObject>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeRO.getList.send();"/>
    </mx:HBox>

    <mx:DataGrid dataProvider="{employeeAC}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, use the `toArray()` method of the `mx.utils.ArrayUtil` class to convert it to an Array, as this example shows. If you pass the `toArray()` method an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array. For information about working with `ArrayCollection` objects, see the Flex documentation.

Accessing EJBs and other objects in JNDI

Access Enterprise JavaBeans (EJBs) and other objects stored in the Java Naming and Directory Interface (JNDI) by calling methods on a destination that is a service facade class that looks up an object in JNDI and calls its methods.

You can use stateless or stateful objects to call the methods of Enterprise JavaBeans and other objects that use JNDI. For an EJB, you can call a service facade class that returns the EJB object from JNDI and calls a method on the EJB.

In your Java class, you use the standard Java coding pattern, in which you create an initial context and perform a JNDI lookup. For an EJB, you also use the standard coding pattern in which your class contains methods that call the EJB home object's `create()` method and the resulting business methods of the EJB.

The following example uses a method called `getHelloData()` on a facade class destination:

```
<mx:RemoteObject id="Hello" destination="roDest">
    <mx:method name="getHelloData"/>
</mx:RemoteObject>
```

On the Java side, the `getHelloData()` method could easily encapsulate everything necessary to call a business method on an EJB. The Java method in the following example performs the following actions:

- Creates new initial context for calling the EJB
- Performs a JNDI lookup that gets an EJB home object
- Calls the EJB home object's `create()` method
- Calls the `sayHello()` method of the EJB

```
public void getHelloData() {
    try
    {
        InitialContext ctx = new InitialContext();
        Object obj = ctx.lookup("/Hello");
        HelloHome ejbHome = (HelloHome)
            PortableRemoteObject.narrow(obj, HelloHome.class);
        HelloObject ejbObject = ejbHome.create();
        String message = ejbObject.sayHello();
    }
    catch (Exception e);
}
```

Calling Remoting Service destinations from Flash or Java applications

The NetConnection API of Flash Player provides a way to call Remoting Service destinations from a standard (non-Flex) Flash application or from ActionScript in a Flex application if desired. The AMFConnection API in LiveCycle Data Services gives you a Java API patterned on the NetConnection API but for calling Remoting Service destinations from a Java application. You can use either of these APIs with LiveCycle Data Services, BlazeDS, or third-party remoting implementations.

Call a destination from a Flash application

You can use the Flash Player `flash.net.NetConnection` API to call a Remoting Service destination from a Flash application. You use the `NetConnection.connect()` method to connect to a destination and the `NetConnection.call()` method to call the service. The following MXML code example shows this way of making Remoting Service calls with `NetConnection` instead of `RemoteObject`:

```
<?xml version="1.0"?>
<!- ds\rpc\NetCon.mxml -->
&ltmx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%"
    creationComplete="creationCompleteHandler();">
    <mx:Script>
        <![CDATA[
            import flash.net.NetConnection;
            import flash.net.ObjectEncoding;
            import flash.net.Responder;
            private var nc:NetConnection;
            private function creationCompleteHandler():void {
                // Create the connection.
                nc = new NetConnection();
                nc.objectEncoding = ObjectEncoding.AMF0;
                // Connect to the remote URL.
                nc.connect("http://[server]:[port]/yourapp/messagebroker/amf");
            }
            private function echo():void {
                // Call the echo method of a destination on the server named remoting_AMF.
                nc.call( "remoting_AMF.echo", new Responder( resultHandler, faultHandler ),
                    ti.text );
            }
            private function resultHandler(result:Object):void {
                ta.text += "Server responded: " + result + "\n";
            }
            private function faultHandler(fault:Object):void {
                ta.text += "Received fault: " + fault + "\n";
            }
        ]]>
    </mx:Script>
    <mx:Panel id="mainPanel" height="100%" width="100%">
        <mx:HBox>
            <mx:Label text="Enter a text for the server to echo"/>
            <mx:TextInput id="ti" text="Hello World!" />
            <mx:Button label="Send" click="echo()"/>
            <mx:Button label="Clear" click='ta.text = ""'/>
        </mx:HBox>
        <mx:TextArea id="ta" width="100%" height="100%"/>
    </mx:Panel>
</mx:Application>
```

Call a destination from a Java application

The AMFConnection API is a Java client API in the flex-messaging-core.jar file that makes it possible to work with Remoting Service destinations from a Java application. To compile Java classes that use the AMFConnection API, you must have both the flex-messaging-core.jar and flex-messaging-common.jar files in your class path.

The AMFConnection API is similar to the Flash Player flash.net.NetConnection API, but uses typical Java coding pattern rather than ActionScript coding pattern. The classes are in the flex.messaging.io.amf.client* package in the flex-messaging-amf.jar file. The primary class is the AMFConnection class. You connect to remote URLs with the AMFConnection.connect() method and call the service with the AMFConnection.call() method. You catch ClientStatusException and ServerStatusException exceptions when there are errors. Here is an example of how you can use AMFConnection to call a Remoting Service destination from a method in a Java class:

```
public void callRemoting()
{
    // Create the AMF connection.
    AMFConnection amfConnection = new AMFConnection();

    // Connect to the remote URL.
    String url = "http://[server]:[port]/yourapp/messagebroker/amf";
    try
    {
        amfConnection.connect(url);
    }
    catch (ClientStatusException cse)
    {
        System.out.println(cse);
        return;
    }

    // Make a remoting call and retrieve the result.
    try
    {
        // Call the echo method of a destination on the server named remoting_AMF.

        Object result = amfConnection.call("remoting_AMF.echo", "echo me1");
    }
    catch (ClientStatusException cse)
    {
        System.out.println(cse);
    }
    catch (ServerStatusException sse)
    {
        System.out.println(sse);
    }
    // Close the connection.
    amfConnection.close();
}
```

The AMFConnection API automatically handles cookies similarly to the way in which web browsers do, so there is no need for custom cookie handling.

Handling service events

Calls to a remote service are asynchronous. After you invoke an asynchronous call, your application does not wait for the result, but continues to execute. Therefore, you typically use events to signal that the service call has completed.

When a service call completes, the WebService and HTTPService components dispatch one of the following events:

- A *result event* indicates that the result is available. A `result` event generates an `mx.rpc.events.ResultEvent` object. You can use the `result` property of the `ResultEvent` object to access the data returned by the service call.
- A *fault event* indicates that an error occurred. A `fault` event generates an `mx.rpc.events.FaultEvent` object. You can use the `fault` property of the `FaultEvent` object to access information about the failure.

You can also handle the `invoke` event, which is broadcast when an RPC component makes the service call. This event is useful if operations are queued and invoked at a later time.

You can handle these events at three different levels in your application:

- Handle events at the component level. Specify event handlers for the `fault` and `result` events for the component. By default, these event handlers are invoked for all service calls.
- For WebService and RemoteObject components, handle events at the operation level. These event handlers override any event handler specified at the component level. The HTTPService component does not support multiple operations, so you use this technique only with the WebService component.
- Handle them at the call level. The `send()` method returns an `AsyncToken` object. You can assign event handlers to the `AsyncToken` object to define event handlers for a specific service call.

Handling events at the component level

Handle events at the component level by using the `fault` and `result` properties of the component to specify the event handlers, as the following example shows:

```
<?xml version="1.0"?>
<!!-- ds\rpc\RPCIntroExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function handleResult(event:ResultEvent):void {
                // Handle result by populating the DataGrid control.
                // The operation returns an Array containing product ID, name, and price.
                myDGdataProvider=event.result;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!!-- Define a WebService component and connect to a service destination. -->
    <mx:WebService
```

```
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"
        result="handleResult(event);"
        fault="handleFault(event);"/>

        <!-- Call the getProducts() operation of the web service.
            The operation takes no parameters. -->
        <mx:Button label="Get Data" click="adbe_news.getProducts();"/>

        <!-- Define a DataGrid control to display the results of the web service. -->
        <mx:DataGrid id="myDG" width="100%" height="100%">
            <mx:columns>
                <mx:DataGridColumn dataField="productId" headerText="Product Id"/>
                <mx:DataGridColumn dataField="name" headerText="Name"/>
                <mx:DataGridColumn dataField="price" headerText="Price"/>
            </mx:columns>
        </mx:DataGrid>

    </mx:Application>
```

In this example, all operations that you invoke through the WebService component use the same event handlers.

Handling events at the operation level for the WebService and RemoteObject component

For the WebService component, you can handle events at the operation level. These event handlers override any event handler specified at the component level. When you do not specify event listeners for an operation, the events are passed to the component level.

In the following MXML example, the WebService component defines default event handlers, and the operation specifies its own handlers:

```
<?xml version="1.0"?>
<!-- ds\rpc\RPCResultFaultMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.SOAPFault;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            public function getProductsResult(event:ResultEvent):void {
                // Handle result.
            }

            public function getProductsFault(event:FaultEvent):void {
                // Handle operation fault.
                Alert.show(event.fault.faultString, "Error");
            }

            public function defaultResult(event:ResultEvent):void {
                // Handle result.
            }

            public function defaultFault(event:FaultEvent):void {
```

```
// Handle service fault.
if (event.fault is SOAPFault) {
    var fault:SOAPFault=event.fault as SOAPFault;
    var faultElement:XML=fault.element;
    // You could use E4X to traverse the raw fault element
    // returned in the SOAP envelope.
}
Alert.show(event.fault.faultString, "Error");
}

]]>
</mx:Script>

<mx:WebService id="WeatherService"
    destination="ws-catalog"
    result="defaultResult(event);"
    fault="defaultFault(event);">
    <mx:operation name="getProducts"
        fault="getProductsFault(event);"
        result="getProductsResult(event);">
    </mx:operation>
</mx:WebService>

<mx:Button label="Get Weather"
    click="WeatherService.getProducts.send();"/>
</mx:Application>
```

Handling events at the call level

Sometimes, an application requires different event handlers for calls to the same service. For example, a service call uses one event handler at application startup, and a different handler for calls during application execution.

To define event handlers for a specific service call, you can use the `AsyncToken` object returned by the `send()` method. You then register event handlers on the `AsyncToken` object, rather than on the component, as the following ActionScript code shows:

```
<?xml version="1.0"?>
<!!-- ds\rpc\RPCAsynchEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.rpc.soap.SOAPFault;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;
            import mx.rpc.AsyncToken;
            import mx.rpc.AsyncResponder;

            // Define an instance of the AsyncToken class.
            public var serviceToken:AsyncToken;
            // Call the service, and then
            // assign the event handlers to the AsyncToken object.
            public function setCustomHandlers():void {
                // send() returns an instance of AsyncToken.
                serviceToken = WeatherService.getProducts.send();
                var asynchRes:AsyncResponder =
                    new AsyncResponder(getProductsResult, getProductsFault);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```
        serviceToken.addResponder(asynchRes);
    }

    // Use the token argument to pass additional information to the handler.
    public function getProductsResult(event:ResultEvent, token:Object = null):void {
        // Handle result.
    }

    // Use the token argument to pass additional information to the handler.
    public function getProductsFault(event:FaultEvent, token:Object = null):void {
        // Handle operation fault.
        Alert.show(event.fault.faultString, "Error");
    }

    public function defaultResult(event:ResultEvent):void {
        // Handle result.
    }

    public function defaultFault(event:FaultEvent):void {
        // Handle service fault.
        if (event.fault is SOAPFault) {
            var fault:SOAPFault=event.fault as SOAPFault;
            var faultElement:XML=fault.element;
            // You could use E4X to traverse the raw fault element
            // returned in the SOAP envelope.
        }
        Alert.show(event.fault.faultString, "Error");
    }
}]]>
</mx:Script>

<mx:WebService id="WeatherService"
    destination="ws-catalog"
    result="defaultResult(event);"
    fault="defaultFault(event);">
    <mx:operation name="getProducts"/>
</mx:WebService>

<!-- Call the service using the default event handlers. -->
<mx:Button label="Use Default Handlers"
    click="WeatherService.getProducts.send();"/>

<!-- Call the service using the custom event handlers. -->
<mx:Button label="Use Custom Handlers"
    click="setCustomHandlers();"/>

</mx:Application>
```

Notice that some properties are assigned to the token after the call to the remote service is made. In a multi-threaded language, there would be a race condition where the result comes back before the token is assigned. This situation is not a problem in ActionScript because the remote call cannot be initiated until the currently executing code finishes.

Handling service results

Most service calls return data to the application. The way to access that data depends on which component you use:

- **HTTPService and WebService**

Access the data by using the `HTTPService.lastResult` property, or in a `result` event by using the `ResultEvent.result` property.

The `WebService` component creates an `Operation` object for each operation supported by the associated web service. Access the data by using the `Operation.lastResult` property for the specific operation, or in a `result` event by using the `ResultEvent.result` property.

By default, the `resultFormat` property value of the `HTTPService` component and of the `Operation` class is `object`, and the data that is returned is represented as a simple tree of ActionScript objects. Flex interprets the XML data that a web service or HTTP service returns to appropriately represent base types, such as `String`, `Number`, `Boolean`, and `Date`. To work with strongly typed objects, populate those objects using the object tree that Flex creates.

`WebService` and `HTTPService` components both return anonymous Objects and Arrays that are complex types. If `makeObjectsBindable` is `true`, which it is by default, Objects are wrapped in `mx.utils.ObjectProxy` instances and Arrays are wrapped in `mx.collections.ArrayCollection` instances.

Note: *ColdFusion is not case-sensitive, so it internally represents all of its data in uppercase. Keep in mind case sensitivity when consuming a ColdFusion web service.*

When consuming data from a web service invocation, you can create an instance of a specific class instead of an Object or an Array. If you want Flex to create an instance of a specific class, use an `mx.rpc.xml.SchemaTypeRegistry` object and register a `QName` object with a corresponding ActionScript class. For more information, see “[Customizing web service type mapping](#)” on page 98.

There is strong data type support between ActionScript types and SOAP and Schema types; for information about data serialization for web services, see “[Serialization between ActionScript and web services](#)” on page 91.

- **RemoteObject**

The `RemoteObject` component creates an `Operation` object for each operation supported by the associated Java class. Access the data by using the `Operation.lastResult` property for the specific operation, or in a `result` event by using the `ResultEvent.result` property.

By default, the `resultFormat` property value of the `Operation` class is `object`, and the data that is returned is represented as a simple tree of ActionScript objects. Flex interprets XML data to appropriately represent base types, such as `String`, `Number`, `Boolean`, and `Date`. To work with strongly typed objects, populate those objects using the object tree that Flex creates.

The `RemoteObject` component returns anonymous Objects and Arrays that are complex types. If `makeObjectsBindable` is `true`, which it is by default, Objects are wrapped in `mx.utils.ObjectProxy` instances and Arrays are wrapped in `mx.collections.ArrayCollection` instances.

There is strong data type support between ActionScript and Java; for information about data serialization between ActionScript and Java, see “[Serialization between ActionScript and Java](#)” on page 79.

Processing results in an event handler

Because calls to a remote service are asynchronous, you typically use events to signal that the service call has completed. A `result` event indicates that the result is available. The event object passed to the event handler is of type `ResultEvent`. Use the `result` property of the `ResultEvent` object to access the data returned by the service call, as the following example shows:

```
<?xml version="1.0"?>
<!!-- ds\rpc\RPCResultEvent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            // Handle result by populating the DataGrid control.
            // The operation returns an Array containing product ID, name, and price.
            public function handleResult(event:ResultEvent):void {
                // Make sure that the result can be cast to the correct type.
                if (event.result is ArrayCollection)
                {
                    // Cast the result to the correct type.
                    myDG.dataProvider=event.result as ArrayCollection;
                }
                else
                    myDG.dataProvider = null;
            }

            public function handleFault(event:FaultEvent):void {
                // Handle fault.
                Alert.show(event.fault.faultString, "Fault");
            }
        ]]>
    </mx:Script>

    <!!-- Define a WebService component and connect to a service destination. -->
    <mx:WebService
        id="adbe_news"
        useProxy="true"
        destination="ws-catalog"
        result="handleResult(event);"
        fault="handleFault(event);"/>

    <!!-- Call the getProducts() operation of the web service.
        The operation takes no parameters. -->
    <mx:Button label="Get Data" click="adbe_news.getProducts();"/>

    <!!-- Define a DataGrid control to display the results of the web service. -->
    <mx:DataGrid id="myDG" width="100%" height="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="productId" headerText="Product Id"/>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="price" headerText="Price"/>
        </mx:columns>
    </mx:DataGrid>

</mx:Application>
```

The data type of the `ResultEvent.result` property is Object. Therefore, the event handler inspects the `ResultEvent.result` property to make sure that it can be cast to the required type. In this example, you want to cast it to ArrayCollection so that you can use it as the data provider for the DataGrid control. If the result cannot be cast to the ArrayCollection, set the data provider to null.

Binding a result to other objects

Rather than using an event handler, bind the results of an RPC component to the properties of other objects, including user interface components and models. The `lastResult` object contains data from the last successful invocation of the component. Whenever a service request executes, the `lastResult` object is updated and any associated bindings are also updated.

In the following example, two properties of the `Operation.lastResult` object for the `GetWeather()` operation of a WebService component, `CityShortName` and `CurrentTemp`, are bound to the `text` properties of two `TextArea` controls. The `CityShortName` and `CurrentTemp` properties are returned when a user makes a request to the `MyService.GetWeather()` operation and provides a ZIP code as an operation request parameter.

```
<mx:TextArea text="{MyService.GetWeather.lastResult.CityShortName}" />
<mx:TextArea text="{MyService.GetWeather.lastResult.CurrentTemp}" />
```

Binding a result to an ArrayCollection object

You can bind the results to the `source` property of an ArrayCollection object, and use the ArrayCollection API to work with the data. You can then bind the ArrayCollection object to a complex property of a user interface component, such as a List, ComboBox, or DataGrid control.

In the following example, bind an `Operation.lastResult` object, `employeeWS.getList.lastResult`, to the `source` property of an ArrayCollection object. The ArrayCollection object is bound to the `dataProvider` property of a DataGrid control that displays the names, phone numbers, and e-mail addresses of employees.

```
<?xml version="1.0"?>
<!!-- ds\rpc\BindingResultArrayCollection.mxml. Warnings on mx:Object -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" verticalGap="10">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.utils.ArrayUtil;
        ]]>
    </mx:Script>

    <mx:WebService id="employeeWS" destination="employeeWS"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');" >
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:ArrayCollection id="ac"
        source="{ArrayUtil.toArray(employeeWS.getList.lastResult)}" />

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">

```

```
<mx:dataProvider>
<mx:ArrayCollection>
<mx:source>
<mx:Object label="Engineering" data="ENG"/>
<mx:Object label="Product Management" data="PM"/>
<mx:Object label="Marketing" data="MKT"/>
</mx:source>
</mx:ArrayCollection>
</mx:dataProvider>
</mx:ComboBox>
<mx:Button label="Get Employee List" click="employeeWS.getList.send();"/>
</mx:HBox>

<mx:DataGrid dataProvider="{ac}" width="100%">
<mx:columns>
<mx:DataGridColumn dataField="name" headerText="Name"/>
<mx:DataGridColumn dataField="phone" headerText="Phone"/>
<mx:DataGridColumn dataField="email" headerText="Email"/>
</mx:columns>
</mx:DataGrid>
</mx:Application>
```

If you are unsure whether the result of a service call contains an Array or an individual object, you can use the `toArray()` method of the `mx.utils.ArrayUtil` class to convert it to an Array. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass an Array to the method, it returns the same Array.

For information about working with `ArrayCollection` objects, see the Flex documentation.

Binding a result to an `XMLListCollection` object

You can bind the results to an `XMLListCollection` object when the `resultFormat` property is set to `e4x`. When using an `XMLListCollection` object, you can use ECMAScript for XML (E4X) expressions to work with the data. You can then bind the `XMLListCollection` object to a complex property of a user interface component, such as a `List`, `ComboBox`, or `DataGrid` control.

In the following example, bind an `Operation.lastResult` object, `employeeWS.getList.lastResult`, to the `source` property of an `XMLListCollection` object. The `XMLListCollection` object is bound to the `dataProvider` property of a `DataGrid` control that displays the names, phone numbers, and e-mail addresses of employees.

Note: To bind service results to an `XMLListCollection`, set the `resultFormat` property of your `HTTPService` or `Operation` object to `e4x`. The default value of this property is `object`.

For more information on handling XML data, see “[Handling results as XML with the E4X result format](#)” on page 181.

```
<?xml version="1.0"?>
<!-- ds\rpc\BindResultXMLListCollection.mxml. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
        ]]>
    </mx:Script>

    <mx:WebService id="employeeWS"
        destination="employeeWS"
        showBusyCursor="true"
        fault="Alert.show(event.fault.faultString, 'Error');">
        <mx:operation name="getList" resultFormat="e4x">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>

    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:ArrayCollection>
                    <mx:source>
                        <mx:Object label="Engineering" data="ENG"/>
                        <mx:Object label="Product Management" data="PM"/>
                        <mx:Object label="Marketing" data="MKT"/>
                    </mx:source>
                </mx:ArrayCollection>
            </mx:dataProvider>
        </mx:ComboBox>

        <mx:Button label="Get Employee List"
            click="employeeWS.getList.send();"/>
    </mx:HBox>

    <mx:XMLListCollection id="xc"
        source="{employeeWS.getList.lastResult}"/>

    <mx:DataGrid dataProvider="{xc}" width="100%">
        <mx:columns>
            <mx:DataGridColumn dataField="name" headerText="Name"/>
            <mx:DataGridColumn dataField="phone" headerText="Phone"/>
            <mx:DataGridColumn dataField="email" headerText="Email"/>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

For information about working with XMLListCollection objects, see the Flex documentation.

Handling results as XML with the E4X result format

You can set the `resultFormat` property value of `HTTPService` components and `WebService` components to `e4x` to specify that the returned data is of type XML. Using a `resultFormat` of `e4x` is the preferred way to work with XML. You can also set the `resultFormat` property to `xml` to specify that the returned data is of type `flash.xml.XMLNode`, which is a legacy object for working with XML. Also, you can set the `resultFormat` property of `HTTPService` components to `flashvars` or `text` to create results as ActionScript objects that contain name-value pairs or as raw text, respectively. For more information, see *Adobe LiveCycle ActionScript Reference*.

When working with web service results that contain .NET `DataSets` or `DataTables`, it is best to set the `resultFormat` property to `object` to take advantage of specialized result handling for these data types. For more information, see “[Handling web service results that contain .NET DataSets or DataTables](#)” on page 184.

Note: If you want to use E4X syntax on service results, set the `resultFormat` property of your `HTTPService` or `WebService` component to `e4x`. The default value is `object`.

When you set the `resultFormat` property of a `WebService` operation to `e4x`, you sometimes have to handle namespace information contained in the body of the SOAP envelope that the web service returns. The following example shows part of a SOAP body that contains namespace information. This data was returned by a web service that retrieves stock quotes. The namespace information is in boldface text.

```
<soap:Body>
  <GetQuoteResponse xmlns="http://ws.invesbot.com/">
    <GetQuoteResult>
      <StockQuote xmlns="">
        <Symbol>ADBE</Symbol>
        <Company>ADOBE SYSTEMS INC</Company>
        <Price>&lt;big&gt;&lt;b&gt;35.90&lt;/b&gt;&lt;/big&gt;</Price>
      </StockQuote>
    </GetQuoteResult>
  </GetQuoteResponse>
  ...
</soap:Body>
```

This `soap:Body` tag contains namespace information. Therefore, if you set the `resultFormat` property of the `WebService` operation to `e4x`, create a namespace object for the `http://ws.invesbot.com/namespace`. The following example shows an application that does that:

```
<?xml version="1.0"?>
<!- ds\rpc\WebServiceE4XResult1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns="* ">

<mx:Script>
<! [CDATA[
import mx.controls.Alert;

private namespace invesbot = "http://ws.invesbot.com/";
use namespace invesbot;
]]>
</mx:Script>

<mx:WebService
id="WS"
destination="stockservice" useProxy="true"
fault="Alert.show(event.fault.faultString, 'Error');">
<mx:operation name="GetQuote" resultFormat="e4x">
<mx:request>
<symbol>ADBE</symbol>
</mx:request>
</mx:operation>
</mx:WebService>

<mx:HBox>
<mx:Button label="Get Quote" click="WS.GetQuote.send() ;"/>
<mx:Text text="{WS.GetQuote.lastResult.GetQuoteResult.StockQuote.Price}" />
</mx:HBox>
</mx:Application>
```

Optionally, you can create a variable for a namespace and access it in a binding to the service result, as the following example shows:

```
<?xml version="1.0"?>
<! -- ds\rpc\WebServiceE4XResult2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns="* ">

<mx:Script>
<! [CDATA[
    import mx.controls.Alert;

    public var invesbot:Namespace =
        new Namespace("http://ws.invesbot.com/");
]]>
</mx:Script>

<mx:WebService
    id="WS"
    destination="stockservice" useProxy="true"
    fault="Alert.show(event.fault.faultString, 'Error');">
    <mx:operation name="GetQuote" resultFormat="e4x">
        <mx:request>
            <symbol>ADBE</symbol>
        </mx:request>
    </mx:operation>
</mx:WebService>

<mx:HBox>
    <mx:Button label="Get Quote" click="WS.GetQuote.send()"/>
    <mx:Text text="{WS.GetQuote.lastResult.invesbot::GetQuoteResult.StockQuote.Price}"/>
</mx:HBox>
</mx:Application>
```

You use E4X syntax to access elements and attributes of the XML that is returned in a `lastResult` object. You use different syntax, depending on whether there is a namespace or namespaces declared in the XML.

No namespace specified

The following example shows how to get an element or attribute value when no namespace is specified on the element or attribute:

```
var attributes:XMLList = XML(event.result).Description.value;
```

The previous code returns `xxx` for the following XML document:

```
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <Description>
        <value>xxx</value>
    </Description>
</RDF>
```

Any namespace specified

The following example shows how to get an element or attribute value when any namespace is specified on the element or attribute:

```
var attributes:XMLList = XML(event.result).*::Description.*::value;
```

The previous code returns `xxx` for either one of the following XML documents:

XML document one:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description>
    <rdf:value>xxx</rdf:value>
  </rdf:Description>
</rdf:RDF>
```

XML document two:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:cm="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <cm:Description>
    <rdf:value>xxx</rdf:value>
  </cm:Description>
</rdf:RDF>
```

Specific namespace specified

The following example shows how to get an element or attribute value when the declared rdf namespace is specified on the element or attribute:

```
var rdf:Namespace = new Namespace("http://www.w3.org/1999/02/22-rdf-syntax-ns#");
var attributes:XMLElementList = XML(event.result).rdf::Description.rdf::value;
```

The previous code returns xxx for the following XML document:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description>
    <rdf:value>xxx</rdf:value>
    <nsX:value>yyy</nsX:value>
  </rdf:Description>
</rdf:RDF>
```

The following example shows an alternate way to get an element or attribute value when the declared rdf namespace is specified on the element or attribute:

```
namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
use namespace rdf;
var attributes:XMLElementList = XML(event.result).rdf::Description.rdf::value;
```

The previous code also returns xxx for the following XML document:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description>
    <rdf:value>xxx</rdf:value>
  </rdf:Description>
</rdf:RDF>
```

Handling web service results that contain .NET DataSets or DataTables

Web services written with the Microsoft .NET Framework can return special .NET DataSet or DataTable objects to the client. A .NET web service provides a basic WSDL document without information about the type of data that it manipulates. When the web service returns a DataSet or a DataTable, data type information is embedded in an XML Schema element in the SOAP message to specify how to process the rest of the message. To best handle results from this type of web service, set the resultFormat property of a Flex WebService operation to object. You can optionally set the resultFormat property of the operation to e4x. However, the XML and E4X formats are inconvenient because you must navigate through the unusual structure of the response and implement workarounds if you want to bind the data to another object.

When you set the `resultFormat` property of a WebService operation to `object`, a `DataTable` or `DataSet` returned from a .NET web service is automatically converted to an object with a `Tables` property, which contains a map of one or more `DataTable` objects. Each `DataTable` object from the `Tables` map contains two properties: `Columns` and `Rows`. The `Rows` property contains the data. The `event.result` object gets the following properties corresponding to `DataSet` and `DataTable` properties in .NET. Arrays of `DataSets` or `DataTables` have the same structures described here, but are nested in a top-level Array on the `result` object.

Property	Description
<code>result.Tables</code>	Map of table names to objects that contain table data.
<code>result.Tables["someTable"].Columns</code>	Array of column names in the order specified in the <code>DataSet</code> or <code>DataTable</code> schema for the table.
<code>result.Tables["someTable"].Rows</code>	Array of objects that represent the data of each table row. For example, { <code>columnName1:value</code> , <code>columnName2:value</code> , <code>columnName3:value</code> }.

The following MXML application populates a `DataGrid` control with `DataTable` data returned from a .NET web service:

```
<?xml version="1.0" encoding="utf-8"?>
<! -- ds\rpc\DataSetDemo.mxml -->
<mx:Application xmlns="*" xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical">

    <mx:Script>
        <! [CDATA[
            import mx.controls.Alert;
            import mx.controls.DataGrid;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            private function onResult(event:ResultEvent):void {
                // A DataTable or DataSet returned from a .NET webservice is
                // automatically converted to an object with a "Tables" property,
                // which contains a map of one or more dataTables.
                if (event.result.Tables != null)
                {
                    // clean up panel from previous calls.
                    dataPanel.removeAllChildren();

                    for each (var table:Object in event.result.Tables)
                    {
                        displayTable(table);
                    }

                    // Alternatively, if a table's name is known beforehand,
                    // it can be accessed using this syntax:
                    var namedTable:Object = event.result.Tables.Customers;
                    //displayTable(namedTable);
                }
            }

            private function displayTable(tbl:Object):void {
                var dg:DataGrid = new DataGrid();
                dataPanel.addChild(dg);
                // Each table object from the "Tables" map contains two properties:
            }
        ]>
    </mx:Script>
</mx:Application>
```

```
// "Columns" and "Rows". "Rows" is where the data is, so we can set
// that as the dataProvider for a DataGrid.
dg.dataProvider = tbl.Rows;
}

private function onFault(event:FaultEvent):void {
    Alert.show(event.fault.toString());
}
]]>
</mx:Script>

<mx:WebService
    id="nwCL"
    wsdl="http://localhost/data/CustomerList.asmx?wsdl"
    result="onResult(event)"
    fault="onFault(event)" />

<mx:Button label="Get Single DataTable"
    click="nwCL.getSingleDataTable() "/>
<mx:Button label="Get MultiTable DataSet"
    click="nwCL.getMultiTableDataSet() "/>
<mx:Panel id="dataPanel"
    width="100%" height="100%"
    title="Data Tables"/>
</mx:Application>
```

The following example shows the .NET C# class that is the back-end web service implementation called by the Flex application. This class uses the Microsoft SQL Server Northwind sample database:

```
<%@ WebService Language="C#" Class="CustomerList" %>
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Services.Description;
using System.Data;
using System.Data.SqlClient;
using System;

public class CustomerList : WebService {
    [WebMethod]
    public DataTable getSingleDataTable() {
        string cnStr = "[Your_Database_Connection_String]";
        string query = "SELECT TOP 10 * FROM Customers";
        SqlConnection cn = new SqlConnection(cnStr);
        cn.Open();
        SqlDataAdapter adpt = new SqlDataAdapter(new SqlCommand(query, cn));
        DataTable dt = new DataTable("Customers");

        adpt.Fill(dt);
        return dt;
    }
}
```

```
[WebMethod]
public DataSet getMultiTableDataSet() {
    string cnStr = "[Your_Database_Connection_String]";
    string query1 = "SELECT TOP 10 CustomerID, CompanyName FROM Customers";
    string query2 = "SELECT TOP 10 OrderID, CustomerID, ShipCity,
    ShipCountry FROM Orders";
    SqlConnection cn = new SqlConnection(cnStr);
    cn.Open();

    SqlDataAdapter adpt = new SqlDataAdapter(new SqlCommand(query1, cn));
    DataSet ds = new DataSet("TwoTableDataSet");
    adpt.Fill(ds, "Customers");

    adpt.SelectCommand = new SqlCommand(query2, cn);
    adpt.Fill(ds, "Orders");

    return ds;
}
```

Handling asynchronous calls to services

Because ActionScript code executes asynchronously, if you allow concurrent calls to a service, ensure that your code handles the results appropriately. By default, making a request to a web service operation that is already executing does not cancel the existing request. In a Flex application in which a service can be called from multiple locations, the service might respond differently in different contexts.

When you design a Flex application, consider whether the application requires disparate data sources, and the number of types of services that the application requires. The answers to these questions help determine the level of abstraction that you provide in the data layer of the application.

In a simple application, user interface components call services directly. In applications that are slightly larger, business objects call services. In still larger applications, business objects interact with service broker objects that call services.

To understand the results of asynchronous service calls to objects in an application, you need a good understanding of scoping in ActionScript. For more information, see the Flex documentation.

Using the Asynchronous Completion Token design pattern

Flex is a service-oriented framework in which code executes asynchronously, therefore, it lends itself well to the Asynchronous Completion Token (ACT) design pattern. This design pattern efficiently dispatches processing within a client in response to the completion of asynchronous operations that the client invokes. For more information, see www.cs.wustl.edu/~schmidt/PDF/ACT.pdf.

When you use the ACT design pattern, you associate application-specific actions and state with responses that indicate the completion of asynchronous operations. For each asynchronous operation, you create an ACT that identifies the actions and state that are required to process the results of the operation. When the result is returned, you can use its ACT to distinguish it from the results of other asynchronous operations. The client uses the ACT to identify the state required to handle the result.

An ACT for a particular asynchronous operation is created before the operation is called. While the operation is executing, the client continues executing. When the service sends a response, the client uses the ACT that is associated with the response to perform the appropriate actions.

When you call a Flex remote object service, web service, or HTTP service, Flex returns an instance of the mx.rpc.AsyncToken class. If you use the default concurrency value of `multiple`, you can use the token returned by the data service's `send()` method to handle the specific results of each concurrent call to the same service.

You can add information to the token when it is returned, and then in a result event listener you can access the token as `event.token`. This situation is an implementation of the ACT design pattern that uses the token of each data service call as an ACT. How you use the ACT design pattern in your own code depends on your requirements. For example, you could attach simple identifiers to individual calls, or more complex objects that perform their own set of functionality, or functions that a central listener calls.

The following example shows a simple implementation of the ACT design pattern. This example uses an HTTP service and attaches a simple variable to the token.

```
<mx:HTTPService id="MyService" destination="httpDest" result="resultHandler(event)"/>

<mx:Script>
    <![CDATA[
        ...
        public function storeCall():void {
            // Create a variable called call to store the instance
            // of the service call that is returned.
            var call:Object = MyService.send();

            // Add a variable to the token that is returned.
            // You can name this variable whatever you want.
            call.marker = "option1";
            ...

        }

        // In a result event listener, execute conditional
        // logic based on the value of call.marker.
        private function resultHandler(event:ResultEvent):void {
            var call:Object = event.token;
            if (call.marker == "option1") {
                //do option 1
            }
            else
            ...
        }
    ]]>
</mx:Script>
```

Making a service call when another call is completed

Another common requirement when using data services is the dependency of one service call on the result of another. Your code must not make the second call until the result of the first call is available. Make the second service call in the result event listener of the first, as the following example shows:

```
<?xml version="1.0"?>
<!- ds\rpc\RPCSecondCall.mxml. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[
            // Call the getForecastWithSalesInput operation with the result of the
            // getCurrentSales operation.
            public function resultHandler(currentsales:String):void {
                ws.setForecastWithSalesInput(currentsales);
                //Or some variation that uses data binding.
            }
        ]]>
    </mx:Script>

    <mx:WebService id="ws" destination="wsDest">
        <mx:operation name="getCurrentSales"
            result="resultHandler(event.result.resultString)"/>
        <mx:operation name="setForecastWithSalesInput"/>
    </mx:WebService>
</mx:Application>
```

Using server-side logging with RPC services

The Proxy Service and Remoting Service log messages through the server-side logging system that is configured in the services-config.xml file. To log messages, use the Service.HTTP or Service.Remoting filter pattern, respectively. For information about server-side logging, see “[Logging](#)” on page 420.

Chapter 5: Message Service

Using the Message Service

The Message Service expands the core messaging framework to add support for publish-subscribe messaging among multiple Flex clients as well as message push from the server to clients. A Flex application uses the client-side messaging API to send messages to, and receive messages from, a destination defined by the server. Messages are sent over a channel and processed by a server endpoint, both of which are protocol-specific.

The Message Service also supports bridging to JMS topics and queues on an embedded or external JMS server by using the JMSAdapter. For more information, see “[Connecting to the Java Message Service \(JMS\)](#)” on page 214.

Introducing the Message Service

You use the Flex client-side API and the corresponding server-side Message Service to create messaging applications. Messaging lets a Flex application connect to a message destination on the server, send messages to the server, and receive messages from other messaging clients. Messages sent to the server are routed to other Flex applications that have subscribed to the same destination.

The server can also push messages to clients on its own. In the server push scenario, the server initiates the message and broadcasts it to a destination. All Flex applications that have subscribed to the destination receive the message. For an example, see the Trader Desktop sample application that ships with LiveCycle Data Services. For more information on the running the sample applications, see “[Running the LiveCycle Data Services sample applications](#)” on page 12.

The Message Service lets separate applications communicate asynchronously as peers by passing messages back and forth through the server. A message defines properties such as a unique identifier, LiveCycle Data Services headers, any custom headers, and a message body. The names of LiveCycle Data Services headers are prefixed by the string "DS".

The most well-known example of the type of application that can use the Message Service is an instant messaging application. In that application, one client sends a message to the server, and the server then routes the message to any subscribed clients. You can create other types of applications to implement broadcast messaging to simultaneously send messages to multiple clients, set up a system to send alert messages, or implement other types of messaging applications.

The following image shows the flow of messages from one Flex client to another. On the Flex client that sends the message, the message is routed over a channel to a destination on the server. The server then routes the message to other Flex clients, which perform any necessary processing of the received message.



Client applications that send messages are called message *producers*. You define a producer in a Flex application by using the Producer component. Client applications that receive messages are called message *consumers*. You define a consumer in a Flex application by using the Consumer component.

Producers send messages to specific destinations on the server. A Consumer component subscribes to a server-side destination, and the server routes any messages sent to that destination to the consumer. In most messaging systems, producers and consumers do not know anything about each other.

A Flex application using the Message Service often contains at least one pair of Producer and Consumer components. This configuration enables each application to send messages to a destination and receive messages that other applications send to that destination.

Types of messaging

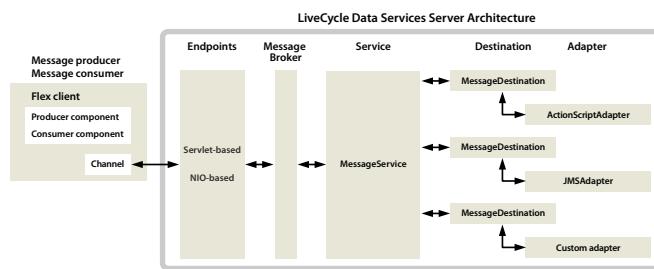
The Message Service supports publish-subscribe messaging. In publish-subscribe messaging, each message can have multiple consumers. You use this type of messaging when you want more than one consumer to receive the same message. Examples of applications that can use publish-subscribe messaging are auction sites, stock quote services, and other applications that require one message to be sent to many subscribers.

The Message Service lets you target messages to specific consumers that are subscribed to the same destination. Consumers can apply a selector expression to the message headers, or Producers and Consumers can add subtopic information to messages to target them. Therefore, even if multiple consumers are subscribed to the same destination, you can target a message to a single consumer or to a subset of all consumers. For more information, see “[Message selectors and subtopics](#)” on page 200.

Note: You can support point-to-point messaging, also known as queue-based messaging, between Flex clients by using the JMSAdapter and bridging to a JMS queue. For more information, see “[Connecting to the Java Message Service \(JMS\)](#)” on page 214.

The Message Service architecture

The components of the Message Service include channels, destinations, adapters, producers, and consumers. The following image shows the messaging architecture:



Channels

Flex applications can access the Message Service over several different message channels. The Flex client tries the channels in the order specified in the configuration files, until an available channel is found or all channels have been tried.

Each channel corresponds to a specific network protocol and has a corresponding server-side endpoint. Use a real-time channel with messaging. Real-time channels include RTMPChannel, AMFChannel and HTTPChannel with polling enabled, and StreamingAMFChannel and StreamingHTTPChannel. The RTMP channel maintains a direct socket connection between the client and the server, so the server can push messages directly to the client with no polling overhead. The AMF and HTTP channels with polling enabled poll the server for new messages when one or more Consumer components on the client have an active subscription.

Message Service

The Message Service maintains a list of message destinations and the clients subscribed to each destination. You configure the Message Service to transport messages over one or more channels, where each channel corresponds to a specific transport protocol.

Destinations

A destination is the server-side code that you connect to using Producer and Consumer components. When you define a destination, you reference one or more message channels that transport messages. You also reference a message adapter or use an adapter that is configured as the default adapter.

Adapters

LiveCycle Data Services provides two adapters to use with the Message Service and lets you create your own custom adapter:

- The ActionScriptAdapter is the server-side code that facilitates messaging when your application uses ActionScript objects only or interacts with another system. The ActionScriptAdapter lets you use messaging with Flex clients as the sole producers and consumers of the messages.
- The JMSAdapter lets you bridge destinations to JMS destinations, topics, or queues on a JMS server so that Flex clients can send messages to and receive messages from the JMS server.
- A custom adapter lets you create an adapter to interact with other messaging implementations, or for situations where you need functionality not provided by either of the standard adapters.

You reference adapters and specify adapter-specific settings in a destination definition of the configuration files.

Message Service configuration

You configure the Message Service by editing the messaging-config.xml file or the services-config.xml file. As a best practice, use the messaging-config.xml file for your configuration to keep it separate from other types of configurations.

Within the services-config.xml file, you specify the channels to use with a destination, set properties of the destination, enable logging, configure security, and specify other properties. For more information, see “[Configuring the Message Service](#)” on page 208.

Working with Producer components

You use the Producer component in a Flex application to enable the application to send messages. You can create Producer components in MXML or ActionScript.

To send a message from a Producer component to a destination, you create an mx.messaging.messages.AsyncMessage object, populate the body of the AsyncMessage object, and then call the `Producer.send()` method. You can create text messages and messages that contain objects.

You can optionally specify `acknowledge` and `fault` event handlers for a Producer component. An `acknowledge` event is dispatched when the Producer receives an acknowledge message to indicate that the destination successfully received a message that the Producer sent. A `fault` event is dispatched when a destination cannot successfully process a message due to a connection-, server-, or application-level failure.

For reference information about the Producer class, see [Adobe LiveCycle ActionScript Reference](#).

Creating a Producer component in MXML

You use the `<mx:Producer>` tag to create a Producer component in MXML. The tag must contain an `id` value. The component typically specifies a destination that is defined in the `messaging-config.xml` file or the `services-config.xml` file. The following code shows an `<mx:Producer>` tag that specifies a destination and `acknowledge` and `fault` event handlers:

```
<?xml version="1.0"?>
<! -- ds\messaging\CreateProducerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.FaultEvent;
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function acknowledgeHandler(event:MessageAckEvent):void {
                // Handle acknowledge message event.
            }

            private function faultHandler(event:MessageFaultEvent):void {
                // Handle message fault event.
            }

            // Compose the message as an instance of AsyncMessage,
            // then use the Producer.send() method to send it.
            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text + ":" + input.text;
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"
        acknowledge="acknowledgeHandler(event);"
        fault="faultHandler(event);"/>

    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage();"/>
</mx:Application>
```

In this example, the Producer component sets the `destination` property to `chat`, which is a destination defined in the `messaging-config.xml` file. You can also specify the destination at run time by setting the `destination` property in your ActionScript code.

To see the data sent by this application, run the application in the section “[Creating a Consumer component in MXML](#)” on page 195 at the same time as you run this application. The application in that section uses a Consumer component to show the data sent by the Producer component.

Creating a Producer component in ActionScript

You can create a Producer component in ActionScript. The following code shows a Producer component that is created in a method in an `<mx:Script>` tag:

```
<?xml version="1.0"?>
<!- ds\messaging\CreateProducerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="createProducer();">

    <mx:Script>
        <! [CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            //Define a variable of type Producer.
            private var producer:Producer;

            // Create the Producer.
            private function createProducer():void {
                producer = new Producer();
                producer.destination = "chat";
                producer.addEventListener(MessageAckEvent.ACKNOWLEDGE, acknowledgeHandler);
                producer.addEventListener(MessageFaultEventFAULT, faultHandler);
            }

            private function acknowledgeHandler(event:MessageAckEvent):void{
                // Handle message acknowledge event.
            }

            private function faultHandler(event:MessageFaultEvent):void{
                // Handle message fault event.
            }
            // Compose the message as an instance of AsyncMessage,
            // then use the Producer.send() method to send it.
            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = userName.text + ":" + input.text;
                producer.send(message);
            }
        ]]>
    </mx:Script>
    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage();"/>
</mx:Application>
```

Resending messages and timing-out requests

A Producer component sends a message once. If the delivery of a message is in doubt, the Producer component dispatches a `fault` event to indicate that it never received an acknowledgment from the destination. When the event is dispatched, the event handler can then make a decision to attempt to resend the message if it is safe to do so.

Note: If a message has side effects, be careful about automatically trying to resend it. If it doesn't have side effects, then you can resend it safely. For example, an HTTP GET operation is read only so it can be safely resent. However, HTTP POST, PUT, and DELETE operations have side effects on the server that make them hard to resend.

Two situations can trigger a fault that indicates delivery is in doubt. It can be triggered when the value of the `Producer.requestTimeout` property is exceeded, or the underlying message channel becomes disconnected before the acknowledgment message is received. The `fault` handler code can detect this scenario by inspecting the `ErrorMessage.faultCode` property of the associated event object for the `ErrorMessage.MESSAGE_DELIVERY_IN_DOUBT` value.

The `Producer.connected` property is set to `true` when the Flex client is connected to the server. The `Producer.send()` method automatically checks this property before attempting to send a message. Two properties control the action of the `Producer` component when it becomes disconnected from the server:

- `reconnectAttempts`

Specifies the number of times the component attempts to reconnect to the server before dispatching a `fault` event. The component makes the specified number of attempts over each available channel. A value of -1 specifies to continue indefinitely and a value of zero disables attempts.

- `reconnectInterval`

Specifies the interval, in milliseconds, between attempts to reconnect. Setting the value to 0 disables reconnection attempts.

Working with Consumer components

You can create Consumer components in MXML or ActionScript. To subscribe to a destination, you call the `Consumer.subscribe()` method.

You can also specify `message` and `fault` event handlers for a Consumer component. A Consumer component broadcasts a `message` event when a message is sent to a destination and the message has been routed to a consumer subscribed to that destination. A `fault` event is broadcast when the channel to which the Consumer component is subscribed cannot establish a connection to the destination, or the subscription request is denied.

For reference information about the `Consumer` class, see the *Adobe LiveCycle ActionScript Reference*.

Creating a Consumer component in MXML

You use the `<mx:Consumer>` tag to create a Consumer component in MXML. The tag must contain an `id` value. It typically specifies a destination that is defined in the server-side `services-config.xml` file.

The following code shows an `<mx:Consumer>` tag that specifies a destination and `acknowledge` and `fault` event handlers:

```
<?xml version="1.0"?>
<!- ds\messaging\CreateConsumerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <! [CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Subscribe to destination.
            private function logon():void {
                consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

            private function faultHandler(event:MessageFaultEvent):void {
                // Handle message fault event.
            }
        ]]>
    </mx:Script>

    <mx:Consumer id="consumer"
        destination="chat"
        message="messageHandler(event);"
        fault="faultHandler(event);"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

You can unsubscribe a Consumer component from a destination by calling the component's `unsubscribe()` method.

To send data to this application, run the application in the section “[Creating a Producer component in MXML](#)” on page 193 at the same time as you run this application. The application in that section uses a Producer component to send data to the Consumer component.

Creating a Consumer component in ActionScript

You can create a Consumer component in ActionScript. The following code shows a Consumer component created in a method in an `<mx:Script>` tag:

```
<?xml version="1.0"?>
<!-- ds\messaging\CreateConsumerAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Create a variable of type Consumer.
            private var consumer:Consumer;

            // Create the Consumer.
            private function logon():void {
                consumer = new Consumer();
                consumer.destination = "chat";
                consumer.addEventListener
                    (MessageEvent.MESSAGE, messageHandler);
                consumer.addEventListener
                    (MessageFaultEvent.FAULT, faultHandler);
                consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

            private function faultHandler(event:MessageFaultEvent):void{
                // Handle message fault event.
            }
        ]]>
    </mx:Script>
    <mx:TextArea id="ta" width="100%" height="100%" />
</mx:Application>
```

Sending and receiving an object in a message

You can send and receive objects as part of a message. The following example sends and receives a message that contains an object:

```
<?xml version="1.0"?>
<!-- ds\messaging\SendObjectMessage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Subscribe to destination.
            private function logon():void {
                consumer.subscribe();
            }

            // Create message from TextInput controls.
            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.body = new Object();
                message.body.uName = userName.text;
                message.body.uInput = input.text;
                message.body.theCollection = ['b','a',3,new Date()];
                producer.send(message);
            }

            // Write received message to TextArea control.
            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text = String(event.message.body.uName) + " , " +
                    String(event.message.body.uInput);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"/>
    <mx:Consumer id="consumer"
        destination="chat"
        message="messageHandler(event);"/>

    <!-- User input controls. -->
    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage() ;"/>

    <!-- Display received message. -->
    <mx:TextArea id="ta"/>
</mx:Application>
```

Handling a network disconnection

Use the `Consumer.subscribed` and `Consumer.connected` properties to monitor the status of the Consumer component. The `connected` property is set to `true` when the Flex client is connected to the server. The `subscribed` property is set to `true` when the Flex client is subscribed to a destination.

Two properties control the action of the Consumer component when the destination becomes unavailable, or the subscription to the destination fails:

- `resubscribeAttempts`

Specifies the number of times the component attempts to resubscribe to the server before dispatching a `fault` event. The component makes the specified number of attempts over each available channel. You can set the `Channel.failoverURIs` property to the URI of a computer to attempt to resubscribe to if the connection is lost. You typically use this property when operating in a clustered environment. For more information, see “[Clustering](#)” on page 442.

A value of `-1` specifies to continue indefinitely, and a value of `0` disables attempts.

- `resubscribeInterval`

Specifies the interval, in milliseconds, between attempts to resubscribe. Setting the value to `0` disables resubscription attempts.

Calling the receive method

Typically, you use a real-time polling or streaming channel with the Consumer component. In both cases, the Consumer receives messages from the server without having to initiate a request. For more information, see “[Channels](#)” on page 191.

You can use a non-real-time channel, such as an AMFChannel with `polling-enabled` set to `false`, with a Consumer component. In that case, call the `Consumer.receive()` method directly to initiate a request to the server to receive any queued messages. Before you call the `receive()` method, call the `subscribe()` method to subscribe to the destination. A `fault` event is broadcast if a failure occurs when the `Consumer.receive()` method is called.

Using a pair of Producer and Consumer components in an application

A Flex application often contains at least one pair of Producer and Consumer components. This configuration enables each application to send messages to a destination and receive messages that other applications send to that destination.

To act as a pair, Producer and Consumer components in an application must use the same message destination. Producer component instances send messages to a destination and Consumer component instances receive messages from that destination.

The following code shows a simple chat application that contains a pair of Producer and Consumer components. The user types messages in a TextInput control; the Producer component sends the message when the user presses the keyboard Enter key or clicks the Button control labeled Send. The user views messages from other users in the TextArea control. If you open this application in two browser windows, any message sent by one instance of the application appears in both.

```
<?xml version="1.0"?>
<!- ds\messaging\ProducerConsumer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <! [CDATA[

            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            // Subscribe to destination.
            private function logon():void {
                consumer.subscribe();
            }

            // Write received message to TextArea control.
            private function messageHandler(event: MessageEvent):void {
                ta.text += event.message.body + "\n";
            }

            // Compose the message as an instance of AsyncMessage,
            // then use the Producer.send() method to send it.
            private function sendMessage():void {
                var message: AsyncMessage = new AsyncMessage();
                message.body = userName.text + ": " + msg.text;
                producer.send(message);
                msg.text = "";
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="chat"/>
    <mx:Consumer id="consumer" destination="chat"
        message="messageHandler(event)"/>

    <mx:TextArea id="ta" width="100%" height="100%"/>

    <mx:TextInput id="userName" width="100%"/>
    <mx:TextInput id="msg" width="100%"/>
    <mx:Button label="Send"
        click="sendMessage();"/>
</mx:Application>
```

Message selectors and subtopics

The Message Service provides functionality for Producer components to add information to message headers and to add subtopic information. Consumer components can then specify filtering criteria based on this information so that only messages that meet the filtering criteria are received by the consumer.

The Consumer component sends the filtering criteria to the server when the Consumer calls the `subscribe()` method. Therefore, while the Consumer component defines the filtering criteria, the actual filtering is done on the server before a message is sent to the consumer.

Note: Filter messages based on message headers or subtopics. However, do not filter messages using both techniques at the same time.

Using selectors

A Producer component can include extra information in a message in the form of message headers. A Consumer component then uses the `selector` property to filter messages based on message header values.

Use the `AsyncMessage.headers` property of the message to specify the message headers. The headers are contained in an associative Array where the key is the header name and the value is either a String or a number.

Note: Do not start message header names with the text "JMS" or "DS". These prefixes are reserved.

The following code adds a message header called `prop1` and sets its value:

```
<?xml version="1.0"?>
<! -- ds\messaging\SendMessageHeader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function sendMessage():void {
                var message:AsyncMessage = new AsyncMessage();
                message.headers = new Array();
                message.headers["prop1"] = 5;
                message.body = input.text;
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"/>

    <mx:TextInput id="userName"/>
    <mx:TextInput id="input"/>
    <mx:Button label="Send"
        click="sendMessage() ; "/>
</mx:Application>
```

To filter messages based on message headers, use the `Consumer.selector` property to specify a message selector. A message selector is a String that contains a SQL conditional expression based on the SQL92 conditional expression syntax. The Consumer component receives only messages with headers that match the selector criteria.

The following code sets the `Consumer.selector` property so that the Consumer only receives messages where the value of `prop1` in the message header is greater than 4:

```
<?xml version="1.0"?>
<!- ds\messaging\CreateConsumerMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"

creationComplete="logon() ;">

<mx:Script>
<! [CDATA[
    import mx.messaging.*;
    import mx.messaging.messages.*;
    import mx.messaging.events.*;

    // Subscribe to destination.
    private function logon():void {
        consumer.subscribe();
    }

    // Write received message to TextArea control.
    private function messageHandler(event:MessageEvent):void {
        // Handle message event.
        ta.text += event.message.body + "\n";
    }
]]>
</mx:Script>

<mx:Consumer id="consumer"
    destination="chat"
    selector="prop1 > 4"
    message="messageHandler(event);"/>
<mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

Note: For advanced messaging scenarios, you can use the `mx.messaging.MultiTopicConsumer` and `mx.messaging.MultiTopicProducer` classes.

If you run the previous two applications at the same time, the Consumer receives messages from the Producer because the Producer sets the value of `prop1` to 5.

Using subtopics

A Producer can send a message to a specific category or categories, called subtopics, within a destination. You then configure a Consumer component to receive only messages assigned to a specific subtopic or subtopics.

Note: You cannot use subtopics with a JMS destination. However, you can use message headers and Consumer selector expressions to achieve similar functionality when using JMS. For more information, see “[Using selectors](#)” on page 201.

In a Producer component, use the `subtopic` property to assign a subtopic to messages. Define a subtopic as a dot (.) delimited String, in the form:

```
mainToken [.secondaryToken] [.additionalToken] [...]
```

For example, you can define a subtopic in the form "chat", "chat.fds", or "chat.fds.newton". The dot (.) delimiter is the default; use the `<subtopic-separator>` property in the configuration file to set a different delimiter. For more information, see “[Setting server properties in the destination](#)” on page 210.

In the Consumer component, use the `subtopic` property to define the subtopic that a message must be sent to for it to be received. You can specify a literal String value for the `subtopic` property. Use the wildcard character (*) in the `Consumer.subtopic` property to receive messages from more than one subtopic.

The Message Service supports single-token wildcard characters (*) in the subtopic String. If the wildcard character is the last character in the String, it matches any tokens in that position or in any subsequent position. For example, the Consumer component specifies the subtopic as "foo.*". It matches the subtopics "foo.bar" and "foo.baz", and also "foo.bar.aaa" and "foo.bar.bbb.ccc".

If the wildcard character is in any position other than the last position, it only matches a token at that position. For example, a wildcard character in the second position matches any tokens in the second position of a subtopic value, but it does not apply to multiple tokens. Therefore, if the Consumer component specifies the subtopic as "foo.*.baz", it matches the subtopics "foo.bar.baz" and "foo.aaa.baz", but not "foo.bar.cookie".

You can use the optional `disallow-wildcard-subtopics` element in the `server` section of a messaging destination to specify whether wildcard characters (*) to receive messages from more than one subtopic are disallowed. The default value is `false`.

To send a message from a Producer component to a destination and a subtopic, set the `destination` and `subtopic` properties, and then call the `send()` method, as the following example shows:

```
<?xml version="1.0"?>
<!- ds\messaging\Subtopic1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <! [CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function useSubtopic():void {
                var message:AsyncMessage = new AsyncMessage();
                producer.subtopic = "chat.fds.newton";
                message.body = "A subtopic message";
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer"
        destination="chat"/>

    <mx:Button label="Send Data"
        click="useSubtopic();"/>
</mx:Application>
```

To subscribe to a destination and a subtopic with a Consumer component, set the `destination` and `subtopic` properties and then call the `subscribe()` method, as the following example shows. This example uses a wildcard character (*) to receive all messages sent to all subtopics under the `chat.fds` subtopic.

```
<?xml version="1.0"?>
<!- ds\messaging\Subtopic2.mxml -->
&ltmx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="logon();">

    <mx:Script>
        <![CDATA[
            import mx.messaging.*;
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function messageHandler(event:MessageEvent):void {
                // Handle message event.
                ta.text += event.message.body + "\n";
            }

            private function logon():void {
                consumer.subtopic = "chat.fds.*";
                consumer.subscribe();
            }
        ]]>
    </mx:Script>

    <mx:Consumer id="consumer"
        destination="chat"
        message="messageHandler(event);"/>
    <mx:TextArea id="ta" width="100%" height="100%"/>
</mx:Application>
```

To allow subtopics for a destination, set the `allow-subtopics` element to `true` in the destination definition in the `messaging-config.xml` file. The `subtopic-separator` element is optional and lets you change the separator character; the default value is `". "` (period).

```
<destination id="chat">
    <properties>
        <network>
            <subscription-timeout-minutes>0</subscription-timeout-minutes>
        </network>
        <server>
            <message-time-to-live>0</message-time-to-live>
            <allow-subtopics>true</allow-subtopics>
            <subtopic-separator>.</subtopic-separator>
        </server>
    </properties>
    <channels>
        <channel ref="my-rtmp"/>
    </channels>
</destination>
```

If you run the previous two applications at the same time, the Consumer receives messages from the Producer.

For more information on configuration, see “[Configuring the Message Service](#)” on page 208.

Multitopic producers and consumers

The MultiTopicProducer component sends messages to a destination with zero or more subtopics. It is like the standard Producer component but can direct the message to any consumer that is subscribing to any one of a number of subtopics. If the consumer is a MultiTopicConsumer component and that consumer has subscribed to more than one subtopic in the list of subtopics used by the MultiTopicProducer, the consumer only receives the message once.

The following code creates a MultiTopicProducer component that sends messages to two subtopics:

```
...
<mx:Script>
...
    function sendMessage():void {
        var producer:MultiTopicProducer = new MultiTopicProducer();
        producer.destination = "NASDAQ";
        var msg:AsyncMessage = new AsyncMessage();
        msg.headers.operation = "UPDATE";
        msg.body = {"SYMBOL":50.00};
        // only send to subscribers to subtopic "myStock1" and "myStock2"
        msg.addSubtopic("myStock1");
        msg.addSubtopic("myStock2");
        producer.send(msg);
    }
...
</mx:Script>
...
```

Unlike the standard Consumer component, the MultiTopicConsumer component lets you register subscriptions for a list of subtopics and selector expressions at the same time from a single message handler. Where the Consumer component has `subtopic` and `selector` properties, this component has an `addSubscription(subtopic, selector)` method you use to add a new subscription to the existing set of subscriptions. Alternatively, you can populate the `subscriptions` property with a list of `SubscriptionInfo` instances that define the subscriptions for the destination.

The following ActionScript code creates a MultiTopicConsumer component that subscribes to two subtopics:

```
...
<mx:Script>
...
    private function initConsumer():void{
        consumer.destination = "NASDAQ";
        consumer.addEventListener(MessageEvent.MESSAGE, messageHandle);
        consumer.addEventListener(MessageFaultEvent.FAULT, faultHandler);
        consumer.addSubscription("myStock1", "operation IN ('BID', 'Ask')");
        consumer.addSubscription("myStock2", "operation IN ('BID', 'Ask')");
        consumer.subscribe();
    }
...
</mx:Script>
...
```

Pushing messages from the server

You can use a Java object on the Livecycle Data Services server to push messages into a Message Service destination. The messages are then distributed to instances of Flex client applications that subscribe to the destination.

To illustrate server push with the Message Service, consider a sample application that includes the following components:

- The feed Message Service destination configured in messaging-config.xml
- The Feed.java Java class that publishes (to the destination) messages that have unique message IDs and contain random numbers.
- The Flex client application that subscribes to the destination and receives messages published to the destination by an instance of the Feed Java class.
- A simple JSP page that creates an instance of the Feed.java class and calls its start() method to start publishing messages to the destination.

Message Service destination

The following example shows the feed Message Service destination:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="message-service"
class="flex.messaging.services.MessageService">
...
    <destination id="feed">
        <properties>
            <network>
                <session-timeout>0</session-timeout>
            </network>
            <server>
                <message-time-to-live>0</message-time-to-live>
                <durable>false</durable>
            </server>
        </properties>
    </destination>
...
</service>
```

Java object that sends messages to destination

The following example shows the Feed.java class:

```
package flex.samples.feed;
import java.util.*;
import flex.messaging.MessageBroker;
import flex.messaging.messages.AsyncMessage;
import flex.messaging.util.UUIDUtils;
public class Feed {
    private static FeedThread thread;
    public Feed() {
    }
    public void start() {
        if (thread == null) {
            thread = new FeedThread();
            thread.start();
        }
    }
    public void stop() {
        thread.running = false;
        thread = null;
    }
    public static class FeedThread extends Thread {
        public boolean running = true;
        public void run() {
            MessageBroker msgBroker = MessageBroker.getMessageBroker(null);
            String clientID = UUIDUtils.createUUID();
            Random random = new Random();
            double initialValue = 35;
            double currentValue = 35;
            double maxChange = initialValue * 0.005;
            while (running) {
                double change = maxChange - random.nextDouble() * maxChange * 2;
                double newValue = currentValue + change;
                if (currentValue < initialValue + initialValue * 0.15
                    && currentValue > initialValue - initialValue * 0.15) {
                    currentValue = newValue;
                } else {
                    currentValue -= change;
                }
                AsyncMessage msg = new AsyncMessage();
                msg.setDestination("feed");
                msg.setClientId(clientID);
                msg.setMessageId(UUIDUtils.createUUID());
                msg.setTimestamp(System.currentTimeMillis());
                msg.setBody(new Double(currentValue));
                msgBroker.routeMessageToService(msg, null);
                System.out.println("" + currentValue);
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                }
            }
        }
    }
}
```

Flex client that subscribes to destination and displays messages

The following example shows the Flex client, which does the following:

- Subscribes to the destination with a button click.
- Handles incoming messages in the `messageHandler()` event handler.
- Displays each message as it arrives from the server in the "pushedValue" TextInput control.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="#FFFFFF">
    <mx:Script>
        <![CDATA[
            import mx.messaging.messages.IMessage;
            private function messageHandler(message:IMessage) :void
            {
                pushedValue.text = ""+ message.body;
            }
        ]]>
    </mx:Script>
    <mx:Consumer id="consumer" destination="feed"
        message="messageHandler(event.message)"/>
    <mx:Button label="Subscribe to 'feed' destination"
        click="consumer.subscribe()" enabled="={!consumer.subscribed}"/>
    <mx:Button label="Unsubscribe from 'feed' destination"
        click="consumer.unsubscribe()" enabled="{{consumer.subscribed}}"/>
    <mx:TextInput id="pushedValue"/>
</mx:Application>
```

JSP that starts message feed

The following example shows the JSP code that starts the message feed:

```
<%@page import="flex.samples.feed.Feed"%>
<%
    try {
        Feed feed = new Feed();
        feed.start();
        out.println("Feed Started");
    } catch (Exception e) {
        out.println("A problem occured while starting the feed: "+e.getMessage());
    }
%>
```

Configuring the Message Service

The most common tasks that you perform when configuring the Message Service are defining message destinations and applying security to message destinations. Typically, you configure the Message Service in the messaging-config.xml file. The services-config.xml file includes the messaging-config.xml file by reference.

The following example shows a basic Message Service configuration in the messaging-config.xml file. It contains the following information:

- A service definition of the Message Service
- A reference to the MessageService class.
- A definition of the ActionScriptAdapter.

- A destination definition that references two channels. You define channels outside the destination definition.

```
<service id="message-service"
    class="flex.messaging.services.MessageService">
    <adapters>
        <adapter-definition
            id="actionscript"
            class="flex.messaging.services.messaging.adapters.ActionScriptAdapter"
            default="true"/>
        <adapter-definition id="jms"
            class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
    </adapters>
    <destination id="chat-topic">
        <properties>
            <server>
                <message-time-to-live>0</message-time-to-live>
            </server>
        </properties>
        <channels>
            <channel ref="samples-rtmp"/>
            <channel ref="samples-amf-polling"/>
        </channels>
    </destination>
</service>
```

Configuring the adapter

To use a Message Service adapter, such as the ActionScript, JMS, or ColdFusion Event Gateway Adapter, you reference the adapter in a destination definition. If you do not explicitly specify an adapter, the destination uses the default adapter as defined in the `<adapters>` tag. In addition to referencing an adapter, you also set its properties in a destination definition.

The following example shows two adapter definitions: the ActionScriptAdapter and the JMSAdapter. In this example, the ActionScriptAdapter is defined as the default adapter:

```
<service id="message-service"
    class="flex.messaging.services.MessageService">
    ...
    <adapters>
        <adapter-definition
            id="actionscript"
            class="flex.messaging.services.messaging.
            adapters.ActionScriptAdapter" default="true"/>
        <adapter-definition
            id="jms"
            class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
    </adapters>
    ...
    <destination id="chat-topic">
        ...
        <adapter ref="actionscript"/>
        ...
    </destination>
    ...
</service>
```

Since ActionScriptAdapter is defined as the default adapter, you can omit the `<adapter>` tag from the destination definition,

Defining the destination

You perform most of the configuration for the Message Service in the destination definition, including specifying the adapter and channels used by the destination, and any network and server properties.

Setting network properties in the destination

A destination contains a set of properties for defining client-server messaging behavior. The following example shows the network-related properties of a destination:

```
<destination id="chat-topic">
    <properties>
        <network>
            <throttle-inbound policy="ERROR" max-frequency="50"/>
            <throttle-outbound policy="ERROR" max-frequency="500"/>
        </network>
    </properties>
</destination>
```

Message Service destinations use the following network-related properties:

Property	Description
subscription-timeout-minutes	Subscriptions that receive no pushed messages in this time interval, in minutes, are automatically unsubscribed. When the value is set to 0 (zero), subscribers are not forced to unsubscribe automatically. The default value is 0.
throttle-inbound	The <code>max-frequency</code> attribute controls how many messages per second the message destination accepts. The <code>policy</code> attribute indicates what to do when the message limit is reached: <ul style="list-style-type: none">• A <code>policy</code> value of <code>NONE</code> specifies no throttling policy (same as frequency of zero).• A <code>policy</code> value of <code>ERROR</code> specifies that when the frequency is exceeded, throttle the message and send an error to the client.• A <code>policy</code> value of <code>IGNORE</code> specifies that when the frequency is exceeded, throttle the message but don't send an error to the client.
throttle-outbound	The <code>max-frequency</code> attribute controls how many messages per second the server can route to subscribed consumers. The <code>policy</code> attribute indicates what to do when the message limit is reached: <ul style="list-style-type: none">• A <code>policy</code> value of <code>NONE</code> specifies no throttling policy (same as frequency of zero).• A <code>policy</code> value of <code>ERROR</code> specifies that when the frequency is exceeded, throttle the message and send an error to the client.• A <code>policy</code> value of <code>IGNORE</code> specifies that when the frequency is exceeded, throttle the message but don't send an error to the client.

Setting server properties in the destination

A destination contains a set of properties for controlling server-related parameters. The following example shows server-related properties of a destination:

```
<destination id="chat-topic">
  <properties>
    ...
    <server>
      <message-time-to-live>0</message-time-to-live>
    </server>
  </properties>
</destination>
```

Message Service destinations use the following server-related properties:

Property	Description
allow-subtopics	(Optional) The subtopic feature lets you divide the messages that a Producer component sends to a destination into specific categories in the destination. You can configure a Consumer component that subscribes to the destination to receive only messages sent to a specific subtopic or set of subtopics. You use wildcard characters (*) to subscribe for messages from more than one subtopic.
disallow-wildcard-subtopics	(Optional) Specifies whether wildcard characters (*) to receive messages from more than one subtopic are disallowed. The default value is <code>false</code> .
cluster-message-routing	(Optional) Determines whether a destination in an environment that uses software clustering uses <code>server-to-server</code> (default) or <code>broadcast</code> messaging. With <code>server-to-server</code> mode, data messages are routed only to servers with active subscriptions, but subscribe and unsubscribe messages are broadcast across the cluster. With <code>broadcast</code> messaging, all messages are broadcast across the cluster. For more information, see “ Clustering ” on page 442.
message-time-to-live	The number of milliseconds that a message is kept on the server pending delivery before being discarded as undeliverable. A value of 0 means the message is not expired.
send-security-constraint	(Optional) Security constraints apply to the operations performed by the messaging adapter. The <code>send-security-constraint</code> property applies to send operations.
subscribe-security-constraint	(Optional) Security constraints apply to the operations performed by the messaging adapter. The <code>subscribe-security-constraint</code> property applies to subscribe, multi-subscribe, and unsubscribe operations.
subtopic-separator	(Optional) Token that separates a hierarchical subtopic value. For example, for the subtopic ‘foo.bar’ the dot (.) is the subtopic separator. The default value is the dot (.) character.

Referencing message channels in the destination

The following example shows a destination referencing a channel. Because the `samples-rtmp` channel is listed first, it is used first and only if a connection cannot be established does the client attempt to connect over the rest of the channels in order of definition.

```
<destination id="chat-topic">
  ...
  <channels>
    <channel ref="samples-rtmp"/>
    <channel ref="samples-amf-polling"/>
  </channels>
  ...
</destination>
```

For more information about message channels, see “[Channels and endpoints](#)” on page 38.

Applying security to the destination

One way to secure a destination is by using a *security constraint*, which defines the access privileges for the destination. You use a security constraint to authenticate and authorize users before allowing them to access a destination. You can specify whether to use basic or custom authentication, and indicate the roles required for authorization.

Two security properties that you can set for a messaging destination include the following:

- `send-security-constraint`
- `subscribe-security-constraint`

Specifies the security constraint for a Producer component sending a message to the server.

Specifies the security constraint for a Consumer component subscribing to a destination on the server.

You use these properties in a destination definition, as the following example shows:

```
<destination id="chat">
  ...
  <properties>
    <server>
      <send-security-constraint ref="sample-users"/>
      <subscribe-security-constraint ref="sample-users"/>
    </server>
  </properties>
  ...
</destination>
```

In this example, the properties reference the sample-users security constraint defined in the services-config.xml file, which specifies to use custom authentication:

```
<security>
  <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
    <per-client-authentication>false</per-client-authentication>
  </login-command>
  <security-constraint id="basic-read-access">
    <auth-method>Basic</auth-method>
    <roles>
      <role>guests</role>
      <role>accountants</role>
    </roles>
  </security-constraint>
  <security-constraint id="sample-users">
    <auth-method>Custom</auth-method>
    <roles>
      <role>sampleusers</role>
    </roles>
  </security-constraint>
</security>
```

For more information about security, see “[Securing LiveCycle Data Services](#)” on page 429.

Creating a custom Message Service adapter

You can create a custom Message Service adapter for situations where you need functionality not provided by the standard adapters. A Message Service adapter class must extend the `flex.messaging.services.messaging.adapters.MessagingAdapter` class. An adapter calls methods on an instance of a `flex.messaging.MessageService` object. The `MessagingAdapter` and `MessageService` classes are in the `flex-messaging-core.jar` file. Documentation for these classes is included in the public LiveCycle Data Services Javadoc documentation.

The primary method of any Message Service adapter class is the `invoke()` method, which is called when a client sends a message to a destination. In the `invoke()` method, you can include code to send messages to all subscribing clients or to specific clients by evaluating selector statements included with a message.

To send a message to clients, you call the `MessageService.pushMessageToClients()` method in your adapter's `invoke()` method. This method takes a message object as its first parameter. Its second parameter is a Boolean value that indicates whether to evaluate message selector statements. You can call the `MessageService.sendPushMessageFromPeer()` method in your adapter's `invoke()` method to broadcast messages to peer server nodes in a clustered environment.

```
package customclasspackage;
{
    import flex.messaging.services.messaging.adapters.MessagingAdapter;
    import flex.messaging.services.MessageService;
    import flex.messaging.messages.Message;
    import flex.messaging.Destination;

    public class SimpleCustomAdapter extends MessagingAdapter {

        public Object invoke(Message message) {
            MessageService msgService = (MessageService)service;
            msgService.pushMessageToClients(message, true);
            msgService.sendPushMessageFromPeer(message, true);
            return null;
        }
    }
}
```

Optionally, a Message Service adapter can manage its own subscriptions by overriding the `ServiceAdapter.handleSubscriptions()` method and return `true`. You also must override the `ServiceAdapter.manage()` method, which is passed `CommandMessages` for subscribe and unsubscribe operations.

The `ServiceAdapter.getAdapterState()` and `ServiceAdapter.setAdapterState()` methods are for adapters that maintain an in-memory state that must be replicated across a cluster. When an adapter starts up, it gets a copy of that state from another cluster node when another node is running.

To use an adapter class, specify it in an `adapter-definition` element in the `messaging-config.xml` file, as the following example shows:

```
<adapters>
...
<adapter-definition id="cfgateway" class="foo.bar.SampleMessageAdapter"/>
...
</adapters>
```

Optionally, you can implement MBean component management in an adapter. This implementation lets you expose properties to a JMX server that can be used as an administration console. For more information, see “[Monitoring and managing services](#)” on page 427.

Using server-side logging with the Message Service

The Message Service logs messages through the server-side logging system that is configured in the `services-config.xml` file. To log messages, use the `Service.Message`, `Service.Message.*`, or `Service.Message.JMS` filter pattern, depending on which service messages you want to log. For information about server-side logging, see “[Logging](#)” on page 420.

Connecting to the Java Message Service (JMS)

The LiveCycle Data Services Message Service supports bridging LiveCycle Data Services to Java Message Service (JMS) messaging destinations by using the JMS Adapter. The JMS Adapter lets Flex clients publish messages to and consume messages from a JMS server. For more information on the Message Service, see “[Using the Message Service](#)” on page 190.

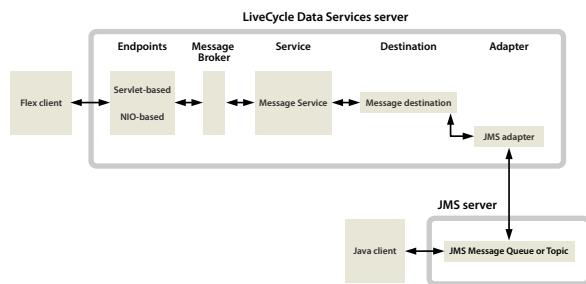
About JMS

Java Message Service (JMS) is a Java API that lets applications create, send, receive, and read messages. Flex applications can then exchange messages with Java client applications.

In a standard configuration of the Message Service, a destination references the ActionScriptAdapter. The ActionScriptAdapter lets you use messaging with Flex clients as the sole producers and consumers of the messages.

You use the JMS Adapter to connect LiveCycle Data Services to JMS topics or queues. The JMS Adapter supports topic-based and queue-based messaging. The JMS Adapter class lets Flex applications participate in existing messaging-oriented middleware (MOM) systems. Therefore, a Java application can publish messages to and respond to messages from Flex applications.

To connect LiveCycle Data Services to a JMS server, you create a destination that references the JMS Adapter. The following image shows LiveCycle Data Services using the JMS Adapter:



The Flex client sends and receives messages through a destination that references the JMS Adapter. The JMS Adapter then connects to the destination on the JMS server. Since the destination is accessible by a Java client, the Java client can exchange messages with the Flex client.

Writing client-side code to use JMS

The Flex client uses the Consumer and Producer components to send and receive messages through the JMS Adapter, just as it would for an application using the ActionScriptAdapter. For example, if the name of a destination that references the JMS Adapter is messaging_JMS_Topic, you reference it from a Producer component as the following example shows:

```
<mx:Producer id="producer"
    destination="messaging_JMS_Topic"
    acknowledge="acknowledgeHandler(event);"
    fault="faultHandler(event);"/>
```

JMS topics and queues

The JMS Adapter supports both JMS topics and JMS queues. The JMS Adapter supports the use of message headers and selectors for JMS topics, but hierarchical topics and subtopics are not supported.

Topics support dynamic client subscribe and unsubscribe, and therefore do not require the same level of administration as JMS queues. When using JMS queues, define a unique queue for each client.

If two Flex clients listen to the same JMS queue, and the JMS server sends a message to the queue, only one of the clients receives the message at a given time. This operation is expected because JMS queues are meant to be consumed by one consumer.

JMS queues are point-to-point, unlike topics which are one-to-many. However, due to the administrative overhead of JMS queues, the JMS Adapter is not the best choice for point-to-point messages between clients. A better choice for point-to-point messaging is to use the ActionScript Adapter in conjunction with message filtering on the client side. For more information, see “[Message selectors and subtopics](#)” on page 200.

Setting up your system to use the JMS Adapter

You can use any JMS server with LiveCycle Data Services that implements the JMS specification. To use the JMS Adapter to connect to a JMS server, you perform several types of configurations, including the following:

- 1 Configure your web application so that it has access to the JMS server. For example, if you are using Tomcat, you might have to add the following Resource definitions to the application to add support for Apache ActiveMQ, which supports JMS version 1.1:

```
<Context docBase="${catalina.home}/../../apps/team" privileged="true"
    antiResourceLocking="false" antiJARLocking="false" reloadable="true">
    <!-- Resourced needed for JMS -->
    <Resource name="jms/flex/TopicConnectionFactory"
        type="org.apache.activemq.ActiveMQConnectionFactory"
        description="JMS Connection Factory"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        brokerURL="vm://localhost"
        brokerName="LocalActiveMQBroker"/>
    <Resource name="jms/topic/flex/simpletopic"
        type="org.apache.activemq.command.ActiveMQTopic"
        description="my Topic"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        physicalName="FlexTopic"/>
    <Resource name="jms/flex/QueueConnectionFactory"
        type="org.apache.activemq.ActiveMQConnectionFactory"
        description="JMS Connection Factory"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        brokerURL="vm://localhost"
        brokerName="LocalActiveMQBroker"/>
    <Resource name="jms/queue/flex/simplequeue"
        type="org.apache.activemq.command.ActiveMQQueue"
        description="my Queue"
        factory="org.apache.activemq.jndi.JNDIReferenceFactory"
        physicalName="FlexQueue"/>
    <Valve className="flex.messaging.security.TomcatValve" />
</Context>
```

The JMS server is often embedded in your J2EE server, but you can interact with a JMS server on a remote computer accessed by using JNDI. For more information, see “[Using a remote JMS provider](#)” on page 219.

- 2 Create a JMS Adapter definition in the messaging-config.xml file:

```
<adapters>
    <adapter-definition id="jms"
        class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
</adapters>
```

For more information on configuring the JMS Adapter, see “[Configure the JMS Adapter](#)” on page 216.

- 3 Create a destination that references the adapter in the messaging-config.xml file:

```
<destination id="messaging_AMF_Poll_JMS_Topic" channels="my-amf-poll">
    <adapter ref="jms"/>
    <properties>
        <jms>
            <connection-factory>
                java:comp/env/jms/flex/TopicConnectionFactory
            </connection-factory>
            <destination-type>Topic</destination-type>
            <destination-jndi-name>
                java:comp/env/jms/topic/flex/simpletopic
            </destination-jndi-name>
            <message-type>javax.jms.TextMessage</message-type>
        </jms>
    </properties>
    <channels>
        <channel ref="samples-rtmp"/>
        <channel ref="samples-amf-polling"/>
    </channels>
</destination>
```

This destination references the first Apache ActiveMQ Resource, which supports topic-based messaging.

Note: Since the channel only defines how the Flex client communicates with the server, you do not have to perform any special channel configuration to use the JMS Adapter.

For more information, see “[Configuring a destination to use the JMS Adapter](#)” on page 216.

- 4 Compile your Flex application against the services-config.xml file, which includes the messaging-config.xml file by reference.

Configuring the Message Service to connect to a JMS Adapter

Typically, you configure the Message Service, including the JMS Adapter, in the messaging-config.xml file.

Configure the JMS Adapter

You configure the JMS Adapter individually for the destinations that use it, as the following example shows:

```
<adapters>
    <adapter-definition id="jms"
        class="flex.messaging.services.messaging.adapters.JMSAdapter"/>
</adapters>
```

Configuring a destination to use the JMS Adapter

You perform most of the configuration of the JMS Adapter in the destination definition. Configure the adapter with the proper JNDI information and JMS ConnectionFactory information to look up the connection factory in JNDI.

The following example shows a destination that uses the JMS Adapter:

```

<destination id="chat-topic-jms">
  <properties>
    ...
    <jms>
      <destination-type>Topic</destination-type>
      <message-type>javax.jms.TextMessage</message-type>
      <connection-factory>jms/flex/TopicConnectionFactory</connection-factory>
      <destination-jndi-name>jms/topic/flex/simpletopic</destination-jndi-name>
      <delivery-mode>NON_PERSISTENT</delivery-mode>
      <message-priority>DEFAULT_PRIORITY</message-priority>
      <preserve-jms-headers>"true"</preserve-jms-headers>
      <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
      <connection-credentials username="sampleuser" password="samplepassword"/>
      <max-producers>1</max-producers>
    </jms>
  </properties>
  ...
  <adapter ref="jms"/>
</destination>

```

The JMS Adapter accepts the following configuration properties. For more specific information about JMS, see the Java Message Service specification or your application server documentation.

Property	Description
acknowledge-mode	Not used with JMS Adapter.
connection-credentials	(Optional) The username and password used while creating the JMS connection for example: <connection-credentials username="sampleuser" password="samplepassword"/> Use only if JMS connection level authentication is being used.
connection-factory	Name of the JMS connection factory in JNDI.
delivery-mode	JMS DeliveryMode for producers. The valid values are PERSISTENT and NON_PERSISTENT. The PERSISTENT mode specifies that all sent messages be stored by the JMS server, and then forwarded to consumers. This configuration adds processing overhead but is necessary for guaranteed delivery. The NON_PERSISTENT mode does not require that messages be stored by the JMS server before forwarding to consumers, so they can be lost if the JMS server fails while processing the message. This setting is suitable for notification messages that do not require guaranteed delivery.
delivery-settings/mode	(Optional) Specifies the message delivery mode used to deliver messages from the JMS server. If you specify <code>async</code> mode, but the application server cannot listen for messages asynchronously (that is <code>javax.jms.MessageConsumer.setMessageListener</code> is restricted), or the application server cannot listen for connection problems asynchronously (for example, <code>javax.jms.Connection.setExceptionListener</code> is restricted), you get a configuration error asking the user to switch to <code>sync</code> mode. The default value is <code>sync</code> .
delivery-settings-sync-receive-interval-millis	(Optional) Default value is 100. The interval of the receive message calls. Only available when the <code>mode</code> value is <code>sync</code> .

Property	Description
delivery-settings/sync-receive-wait-millis	(Optional) Default value is 0 (no wait). Determines how long a JMS proxy waits for a message before returning. Using a high sync-receive-wait-millis value along with a small thread pool can cause messages to back up if many proxied consumers are not receiving a steady flow of messages. Only available when the mode value is sync.
destination-jndi-name	Name of the destination in the JNDI registry.
destination-type	(Optional) Type of messaging that the adapter is performing. Valid values are topic for publish-subscribe messaging and queue for point-to-point messaging. The default value is topic.
initial-context-environment	A set of JNDI properties for configuring the InitialContext used for JNDI lookups of your ConnectionFactory and Destination. Lets you use a remote JNDI server for JMS. For more information, see “Using a remote JMS provider” on page 219.
max-producers	The maximum number of producer proxies that a destination uses when communicating with the JMS server. The default value is 1, which indicates that all clients using the destination share the same connection to the JMS server.
message-priority	JMS priority for messages that producers send. The valid values are DEFAULT_PRIORITY or an integer value indicating the priority. The JMS API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Additionally, clients should consider priorities 0-4 as gradations of normal priority, and priorities 5-9 as gradations of expedited priority.
message-type	Type of message to use when transforming Flex messages into JMS messages. Supported types are javax.jms.TextMessage and javax.jms.ObjectMessage. If the client-side Publisher component sends messages as objects, set the message-type to javax.jms.ObjectMessage.
message-type	The javax.jms.Message type which the adapter uses for this destination. Supported types are javax.jms.TextMessage and javax.jms.ObjectMessage.
preserve-jms-headers	(Optional) Defaults to true. Determines whether the adapter preserves all standard JMS headers from JMS messages to LiveCycle Data Services messages. Every JMS message has a set of standard headers: JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSExpiration, JMSRedelivered, JMSPriority, JMSReplyTo, JMSCorrelationID, and JMSType. The JMS server sets these headers when the message is created and they are passed to LiveCycle Data Services. LiveCycle Data Services converts the JMS message into a LiveCycle Data Services message and sets JMSMessageID and JMSTimestamp on the LiveCycle Data Services message as messageID and timestamp, but the rest of the JMS headers are ignored. Setting the preserve-jms-headers property to true preserves all of the headers.

Configuring a server for the JMS Adapter

When a destination specifies a <destination-type> of topic for the JMS Adapter, you can set the durable property in the server definition. When true, the durable property specifies that messages are saved in a durable message store to ensure that they survive connection outages and reach destination subscribers. The default value is false.

Note: This property does not guarantee durability between Flex clients and the JMS Adapter, but between the JMS Adapter and the JMS server.

The following example sets the durable property to true:

```
<server>
    <durable>true</durable>
</server>
```

Using a remote JMS provider

In many cases, the JMS server is embedded in your J2EE server. However, you can also interact with a JMS server on a remote computer accessed by using JNDI.

You can use JMS on a remote JNDI server by configuring the optional `initial-context-environment` element in the `jms` section of a message destination that uses the JMS Adapter. The `initial-context-environment` element takes property subelements, which in turn take `name` and `value` subelements. To establish the desired JNDI environment, specify the `javax.naming.Context` constant names and corresponding values in the `name` and `value` elements, or specify String literal names and corresponding values.

The bold-faced code in the following example is an `initial-context-environment` configuration:

```
<destination id="chat-topic-jms">
    <properties>
        ...
        <jms>
            <destination-type>Topic</destination-type>
            <message-type>javax.jms.TextMessage</message-type>
            <connection-factory>jms/flex/TopicConnectionFactory</connection-factory>
            <destination-jndi-name>jms/topic/flex/simpletopic</destination-jndi-name>
            <delivery-mode>NON_PERSISTENT</delivery-mode>
            <message-priority>DEFAULT_PRIORITY</message-priority>
            <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>

            <!-- (Optional) JNDI environment. Use when using JMS on a remote JNDI server. -->
            <initial-context-environment>
                <property>
                    <name>Context.SECURITY_PRINCIPAL</name>
                    <value>anonymous</value>
                </property>
                <property>
                    <name>Context.SECURITY_CREDENTIALS</name>
                    <value>anonymous</value>
                </property>
                <property>
                    <name>Context.PROVIDER_URL</name>
                    <value>http://{server.name}:1856</value>
                </property>
                <property>
                    <name>Context.INITIAL_CONTEXT_FACTORY</name>
                    <value>fiorano.jms.runtime.naming.FioranoInitialContextFactory</value>
                </property>
            </initial-context-environment>
        </jms>
    </properties>
    ...
    <adapter ref="jms"/>
</destination>
```

Flex treats `name` element values that begin with the text "Context ." as constants defined by `javax.naming.Context` and verifies that the `Context` class defines the constants that you specify. Some JMS providers also allow custom properties to be set in the initial context. You can specify these properties by using the string literal name and corresponding value that the provider requires. For example, the FioranoMQ JMS provider configures failover to back up servers with the following property:

```
<property>
    <name>BackupConnectURLs</name>
    <value>http://backup-server:1856;http://backup-server-2:1856</value>
</property>
```

If you do not specify the `initial-context-environment` properties in the `jms` section of a destination definition, the default JNDI environment is used. The default JNDI environment is configured in a `jndiprovider.properties` application resource file and or a `jndi.properties` file.

Naming conventions across JNDI providers for topic connection factories and destinations can vary. Depending on your JNDI environment, the `connection-factory` and `destination-jndi-name` elements must correctly reference the target named instances in the directory. Include the client library JAR files for your JMS provider in the WEB-INF/lib directory of your web application, or in another location from which the class loader loads them. Even when using an external JMS provider, LiveCycle Data Services uses the `connection-factory` and `destination-jndi-name` configuration properties to look up the necessary connection factory and destination instances.

J2EE restrictions on JMS

Section 6.6 of the J2EE 1.4 specification limits some of the JMS APIs that can be used in a J2EE environment. The following table lists the restricted APIs that are relevant to the JMS Adapter and the implications of the restrictions.

API	Implication of restriction	Alternative
<code>javax.jms.MessageConsumer.get/setMessageListener</code>	No asynchronous message delivery	Set the <code>mode</code> attribute of the <code>delivery-settings</code> configuration property to <code>sync</code> .
<code>javax.jms.Connection.setExceptionListener</code>	No notification of connection problems	Set the <code>mode</code> attribute of the <code>delivery-settings</code> configuration property to <code>sync</code> .
<code>javax.jms.Connection.setClientID</code>	No durable subscribers	Set the <code>durable</code> configuration property to <code>false</code> .
<code>javax.jms.Connection.stop</code>	Not an important implication	None

The JMS Adapter handles these restrictions by doing the following:

- Providing an explicit choice between synchronous and asynchronous message delivery.
- When asynchronous message delivery is specified and `setMessageListener` is restricted, a clear error message is sent to ask the user to switch to synchronous message delivery.
- When asynchronous message delivery is specified and `setExceptionListener` is restricted, a clear error message is sent to ask the user to switch to synchronous message delivery.
- Providing fine-grained tuning for synchronous message delivery so asynchronous message delivery is not needed.
- Providing clear error messages when durable subscribers cannot be used due to `setClientID` being restricted and asking the user to switch durable setting to `false`.

Chapter 6: Data Management Service

Introducing the Data Management Service

The Adobe LiveCycle Data Services Data Management Service automates data synchronization between Adobe Flex client application and the middle tier. You can build applications that provide data synchronization, on-demand data paging, and occasionally connected application services. Additionally, you can manage large collections of data and nested data relationships, such as one-to-one and many-to-one associations.

With the data modeling features introduced in LiveCycle Data Services 3, you can use model-driven development to take advantage of advanced Data Management Service features without writing Java code or configuring services on the server. For more information, see “[Model-driven applications](#)” on page 326 and “[The Model Assembler](#)” on page 293.

Note: The Data Management Service is not available in BlazeDS.

About the Data Management Service

Why the Data Management Service is useful

The Data Management Service lets you reduce the amount of code required to write a rich Internet application. You can write most of the code in a declarative style in XML. Using the Data Management Service, you can quickly and easily build and maintain applications, and enhance collaboration between programmers and designers. This method of programming also provides features such as on-demand paging, synchronization of data between client, server, and other clients for improved access to information and collaboration. You can also make data available offline. The Data Management Service is built on the LiveCycle Data Services message-based framework, a flexible, efficient and scalable framework that integrates seamlessly with existing J2EE and ColdFusion server environments.

When you use the Data Management Service, changes to data at the Flex client side are automatically batched and sent to the Data Management Service running in your application server. The Data Management Service then passes the changes to your business layer or directly to your persistence layer; you can use data access objects (DAOs) with straight JDBC calls, Hibernate, Java Persistence API (JPA), or any other solution. By using the Data Management Service, you can avoid writing data synchronization code in your Flex client application.

Depending on the type of application you are building, the Data Management Service can save you from writing a great deal of client-side code in the following scenarios:

- Keeping track of all the items created, updated, and deleted by the user at the client side.
- Keeping track of the original value of the data as initially retrieved by the client. The persistence layer often needs the original value to implement optimistic locking.
- Making a series of RPC calls to send changes (creates, updates, deletes) to the middle tier.
- Handling the conflicts that arise during the data synchronization process.

Comparing the Data Management Service and RPC approaches

The Data Management Service uses an approach to data that is fundamentally different from the remote procedure call (RPC) approach. Unlike the RPC approach, the Data Management Service supports automatic and manual synchronization of a common set of data on multiple clients and server-side data resources. The client automatically tracks changes made to these objects and can apply those changes on the server objects. The server can then update any clients viewing these same objects. It also supports offline client-side data persistence.

Note: Flex 4 supports a limited subset of data management features for RPC services on the client. You use these features in conjunction with the data-centric development features in Flash Builder 4. For more information, see *Creating data-driven applications with Flex*.

Flow of data between clients and the server

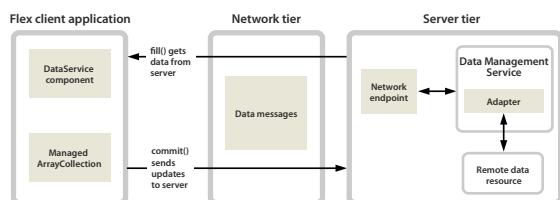
The Data Management Service uses the LiveCycle Data Services message-based framework, which passes data messages between Flex client applications and the Data Management Service. Data Management Service client applications act as both message producers and consumers that send data to a Data Management Service destination and subscribe to the destination to receive data updates.

The primary client-side object you use with the Data Management Service is the DataService component. You can create a DataService component in MXML or ActionScript, and call methods on a Data Management Service destination. The DataService component performs activities such as filling client-side objects with data from remote data resources, committing data changes to the server, and detecting data synchronization conflicts. For more information, see “[Data Management Service clients](#)” on page 226.

Note: If you build model-driven applications that use the Data Management Service, you do not invoke the DataService component directly. Instead, you work with a service wrapper class that uses the DataService component internally. For more information, see “[Model-driven applications](#)” on page 326.

Changes to data on the client are tracked and sent to the Data Management Service destination when you choose to commit them. If successfully applied, they are propagated to other clients.

The following image shows the flow of data between a back-end data resource and an ArrayCollection object that a DataService component manages in a Flex application:



Key concepts of data management

Managed objects

A managed object, also referred to as a *persistent entity*, is an object for which the Data Management Service manages client-server communication. A managed object is represented as a Java object on the server and an ActionScript object in a Flex client.

The Data Management Service can manage both strongly typed objects, which have an explicit class, and weak typed or anonymous objects. In ActionScript, a weak typed object is represented as an instance of the Object class. In Java, it is represented by an object that implements the java.util.Map interface. The Data Management Service uses the standard LiveCycle Data Services serialization mechanism to convert from ActionScript to Java and that mechanism supports mixing strong and weak data types, and performing customized conversions between these classes in a flexible manner.

The Data Management Service must intercept the get and set property operations for its managed instances. This allows it to track changes made to managed objects from your application and to implement on-demand loading features. If you use a weakly typed object, the object is proxied by an instance of mx.data.ManagedObjectProxy, a dynamic object wrapper that performs this logic. The existence of this proxy is usually transparent to your application. However, you cannot use properties with the same names as properties already in the ManagedObjectProxy instance. Avoid using the following property names in your objects:

- dispatcher
- notifiers
- object
- propertyList
- proxyClass
- uid

When you use a strongly typed class in ActionScript, the easiest way to add managed behavior to your object is to use the [Managed] metadata tag at the class level in your ActionScript class.

When using model-driven development and the Model Assembler, managed ActionScript objects are generated automatically; for more information, see “[Model-driven applications](#)” on page 326.

Entities and value objects

Data management distinguishes between two types of objects: *entities* and *value objects*. Entities are objects that have one or more *identity properties* that, when combined, determine a unique fixed identity for the object. For example, a customer record in a database maps to an entity in data management. The entity could have a `customerid` property that maps to a CUSTOMERID primary key column of the database table and that property would be an identity property. For more information, see “[Identity properties](#)” on page 223.

A value object is an object for which the identity is determined by the values of its properties. Examples of value objects include primitive object types such as int and String, and complex objects such as objects that store the parameters to a query.

The Data Management Service only directly manages entities, but it uses value objects. A managed entity is also called a *persistent entity*. Identity properties of entities must be value objects. A property of a managed object is treated as a value object unless the property has a managed association. In that case, the property defines a reference to one or more other managed objects.

Identity properties

When you use data management, all persistent entities must have one or more identity properties that uniquely identify each managed instance. Generally, each identity property maps to a primary key column in a relational database table that the managed object represents.

Identity values can be initially undefined when you create a new entity instance. However, after the server has persisted the entity, the identity values must be fixed for the lifetime of the object. Data management ensures that each client only works with a single version of each entity instance. When the client receives an updated version of an entity instance from the server, it merges those changes into that instance.

For model-driven development, you specify identity properties in id child elements of an entity element. For non-model development, you specify identity properties in destination definitions in a configuration file; for more information, see “[Uniquely identifying data items](#)” on page 249.

Queries

The primary way to fetch data from the server is with a `DataService.fill()` method on the Flex client. The `fill()` method populates an `ArrayCollection` object with the retrieved data. Any parameters in the `fill()` method define a query for the data to retrieve, and the method returns a collection of objects from the server based on that query. A `fill()` method with no parameters retrieves all items.

Parameters of `fill()` method are value objects that form a unique identity for a particular query. A set of fill parameters returns a consistent set of managed entities to the client. The entities act as a key to identify the list of objects. Other clients using the same key get the same set of objects when you use the synchronization features of data management.

There are two ways to implement queries on the server. Using the default approach, you provide a method that takes the set of query parameters from the `fill()` method on the client and returns the entire collection to the client. You can also use the paged-fill approach where your method takes additional start and count parameters. Paging works with either approach. In the first case, paging only pages from the client to the server. In the second case, you page directly from the client to the database.

Modeling relationships between objects

The Data Management Service supports two basic approaches to modeling relationships in your object model. You can manipulate identity properties directly in your object model, or you can model those relationships using associations, which are strongly typed references to objects of the associated type. For destinations configured in the `services-config.xml` file, you declare associations in XML elements in a destination definition. You can declare many-to-one, one-to-one, one-to-many, and many-to-many associations between objects.

When using model-driven development and the Model Assembler, you establish associations in entity elements in a data model; for more information, see “[Model-driven applications](#)” on page 326.

Using identity properties directly in your object model is a fairly simple approach but involves more code for fetching and managing the relationships between objects. Instead of referring to related objects using the dot notation (`a.b[i].c`), you execute queries from a method you call explicitly.

There is no correct approach to how you model objects in every situation, and your choice is largely a matter of style. Queries are more flexible in terms of how you can fetch the data; it is easy to add new parameters to the query for sorting or filtering the query.

Associations manage membership of the query automatically. Using queries sometimes requires more work to make sure that membership changes in the query results are properly and efficiently updated on the clients. With queries, you must execute the query explicitly in ActionScript code, but with association properties you can leverage lazy and load-on-demand behavior so that queries are executed automatically as the application tries to fetch the data.

For more information, see “[Hierarchical data](#)” on page 293.

Destinations and classes

Each DataService component on the client manages the data for a particular destination on the server. A destination is typically created for each type of data that you want to manage. For the most programming flexibility when you have inheritance in your data model, define a destination for each class and set the `item-class` attribute for the destination. One destination can extend another destination to mirror your class hierarchy, making it easier to manage complex object models. If you do not set the `item-class` attribute for a particular destination, that destination can manage arbitrary types of objects that share the same identity property and associations.

Adapters and assemblers

An adapter is responsible for updating the persistent data store in a manner appropriate to the data store type. The Java adapter delegates this responsibility to a class called an assembler class. For more information, see “[Understanding data management adapters](#)” on page 245.

Note: When you build model-driven applications with the Model Assembler, you do not manually code or configure assemblers. A Model Assembler is automatically instantiated for each entity element in a model file when you deploy the model file to the server. For more information, see “[Model-driven applications](#)” on page 326.

Strategies for fetching objects

When you define an explicit association between entities, you have several options for specifying how the client fetches the associated entity object. When the `lazy` property is set to `false` and the `load-on-demand` property is set to `false`, the associated object or objects are fetched when the referencing object is fetched.

If you set the `lazy` property of an association to `true`, the identities of the associated object are fetched when the parent object is fetched. If you set the `load-on-demand` property to `true`, no information for the associated object is fetched when the parent object is fetched. Instead, the property value is fetched the first time the client asks for that value. For a one-to-many, many-to-one, or many-to-many association, you can additionally use the `page-size` property so that when you do fetch the value of the collection, it is retrieved one page at a time.

Note: When you use load-on-demand or paging for a property, the Data Management Service cannot detect conflicts made to that property.

For more information, see “[Data paging](#)” on page 303.

Paging and lazy loading

The Data Management Service provides several ways to improve performance with large sets of data by incrementally paging data from the server to the client and from server to the database. For more information, see “[Data paging](#)” on page 303.

Transactions

By default, LiveCycle Data Services encapsulates client-committed operations in a single J2EE distributed transaction. All client data changes are handled as a unit. If one value cannot be changed, changes to all other values are rolled back. For more information, see “[Using transactions](#)” on page 248.

Conflict resolution

The Data Management Service provides exceptions that let you handle conflicts between versions of data items on the client and server. For more information, see “[Handling data synchronization conflicts](#)” on page 242.

Data Management Service clients

LiveCycle Data Services includes a client-side DataService component that you use in conjunction with the server-side Data Management Service to distribute and synchronize data among multiple client applications. You create client-side Flex applications that can share and synchronize distributed data.

Note: With the data modeling features introduced in LiveCycle Data Services 3, you automatically generate client-side service wrapper classes and supporting classes in Flash Builder instead of using the DataService component directly. For more information, see “[Model-driven applications](#)” on page 326.

For information about configuring the server-side Data Management Service, see “[Data Management Service configuration](#)” on page 244 and “[Standard assemblers](#)” on page 268 and “[Custom assemblers](#)” on page 254.

Creating a Data Management Service client

A Flex client application uses a client-side DataService component to receive data from, and send data to, the server-side Data Management Service. A DataService component can fill a client-side ArrayCollection object with data and manage synchronization of the ArrayCollection object data with the versions of data in other clients and on the server. You can create DataService components in MXML or ActionScript.

A DataService component requires a valid Data Management Service destination. You define destinations in the data-management-config.xml configuration file. For information about Data Management Service destinations, see “[Data Management Service configuration](#)” on page 244.

Creating a DataService component

A DataService component manages the interaction with a server-side Data Management Service destination. You can create a DataService component in MXML or ActionScript.

The following example shows MXML code for creating a DataService component. The DataService component destination property must reference a valid server-side Data Management Service destination.

```
<?xml version="1.0"?>
<! -- ds\datamanagement\DataServiceMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:DataService id="ds" destination="contact"/>
</mx:Application>
```

The following example shows ActionScript code for creating the same DataService component:

```
<?xml version="1.0"?>
<! -- ds\datamanagement\DataServiceAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="initApp();">
    <mx:Script>
        <! [CDATA[
            import mx.data.DataService;
            public var ds:DataService;

            public function initApp():void {
                ds = new DataService("contact");
            }
        ]>
    </mx:Script>
</mx:Application>
```

When you create a DataService component in ActionScript, you must import the mx.data.DataService class and declare a variable of type DataService, for which you set the value to a new DataService object.

Getting a set of data from a destination

When you call a DataService component `fill()` method, you fill an ArrayCollection object with the data from a Data Management Service destination. You can create an ArrayCollection object in MXML or ActionScript. The ArrayCollection API provides a set of methods and properties for manipulating a set of data; for information, see the Flex documentation set.

To release an ArrayCollection object that you filled, you call the DataService component `releaseCollection()` method. If you call the `fill()` method again on the same ArrayCollection object with the same parameters, it fetches a fresh copy of the data. If you call the `fill()` method again with different parameters, it releases the first fill and then fetches the new one.

The first parameter of a `fill()` method is the ArrayCollection to fill. The values of any additional parameters depend on the type of server-side destination that you call.

For example, when you call a destination that uses the Java adapter with a custom assembler, the arguments following the ArrayCollection to fill could be the arguments of a corresponding server-side method that is declared in the destination.

The Data Management destination is responsible for interpreting the query parameters. The HibernateAssembler and SQLAssembler use a common pattern: the first parameter is the name of the query; the second parameter specifies an object that contains the values for the arguments to that query (if any) by using a name-value format, as the following example shows:

```
...
var myFirstName:String = "...";
myService.fill(myCollection, "getByFirstName", {firstName:myFirstName});
...
```

The value of the `myFirstName` variable is substituted into the query expecting a parameter named `firstName`.

For more information, see “[Data Management Service configuration](#)” on page 244.

Note: To improve the speed of your application, you can set the `DataService.indexReferences` property to `false` if you have a small number of fills or references to items managed by a DataService component from association properties of other items.

Populating an ArrayCollection and data provider control with data

To populate a data provider control, such as a DataGrid control, with data, you can use data binding to bind a managed ArrayCollection object to the data provider control `dataProvider` property. When you set the associated DataService component `autoCommit` property to `true`, changes to data in the DataGrid are automatically sent to the Data Management Service destination.

The following example shows an ArrayCollection object that is bound to a DataGrid control `dataProvider` property:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
backgroundColor="#FFFFFF">

<mx:ArrayCollection id="products"/>
<mx:DataService id="ds" destination="inventory"/>

<Product/>

<mx:DataGrid dataProvider="{products}" editable="true" width="100%" height="100%">

<mx:columns>

<mx:DataGridColumn dataField="name" headerText="Name" />

<mx:DataGridColumn dataField="category" headerText="Category" />

<mx:DataGridColumn dataField="price" headerText="Price" />

<mx:DataGridColumn dataField="image" headerText="Image" />

<mx:DataGridColumn dataField="description" headerText="Description" />

</mx:columns>

</mx:DataGrid>

<mx:Button label="Get Data" click="ds.fill(products)"/>
</mx:Application>
```

Sending changes from a managed ArrayCollection object

By default, the `commit()` method of a DataService component is automatically called when data changes in the ArrayCollection object that it manages. You can also call the `commit()` method manually and set a DataService component `autoCommit` property to `false` to allow only manual calls to the `commit()` method. It is important to set `autoCommit` to `false` when you are going to make more than one change in the same frame so that the DataService component can batch those changes and send them in one batch to the destination.

The following example shows a manual update operation on an item in an ArrayCollection object that a DataService component manages:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="initApp();">
    <mx:Script>
        <![CDATA[
            import mx.data.DataService;
            import mx.collections.ArrayCollection;
            import samples.customer.Customer;
            ...

            public function initApp():void {
                var customers:ArrayCollection = new ArrayCollection();
                var customerService:DataService = new DataService("customers");
                customerService.autoCommit = false;
                var customer:Customer = customers.getItemAt(4);
                customer.city = "Oakland";
                customer.name = "CyberTech Enterprises";
                customerService.commit();
            }
            ...
        ]]>
    </mx:Script>
</mx:Application>
```

The DataService component creates a single update message that includes the change to the `customer.name` and `customer.city` properties. This message is sent to the destination when the DataService component's `commit()` method is called.

The following example shows two manual update operations on an item in an ArrayCollection object that a DataService component manages:

```
...
var customer:Customer = customers.getItemAt(4);
var oldName:String = customer.name;
customer.name = "CyberTech Enterprises";
customer.name = oldName;
customerService.commit();
...
```

If the update value is the same as the original value, there is no update operation issued. When there are multiple updates to the same item, only the latest one is issued. If it is same as original value, there is no update issued.

When the value of the `customer.name` property is changed to "CyberTech Enterprises", the DataService component creates a single update message. On the subsequent change back to the old name by using `customer.name = oldName`, the original update message for the customer name is removed. The result is that nothing is sent when the `commit()` method is called.

The following example shows the addition and removal of a customer to an ArrayCollection object that the DataService component manages:

```
<?xml version="1.0"?>
<!!-- fds\datamanagement\AddRemoveItem.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initApp();">

    <mx:Script>
        <! [CDATA[
            import mx.collections.ArrayCollection;
            import mx.data.DataService;
            import samples.customer.Customer;
            public function initApp():void {
                var customers:ArrayCollection = new ArrayCollection();
                var customerService:DataService = new DataService("customers");
                customerService.autoCommit = false;
                customers.addItemAt(new Customer(), 4);

                // Remove the previously added customer.
                customers.removeItemAt(4);
                customerService.commit();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

The DataService component attempts to log two updates that cancel out each other. When the `addItemAt()` method is called, the DataService component creates a create message for the `customers` destination. On the subsequent call to `removeItemAt(4)`, the previous create message is removed. Nothing is sent when the `commit()` method is called.

Working with single data items

The `mx.data.DataService` class has several methods for working with individual data items. The following table describes these methods:

Method	Description
<code>createItem()</code>	Lets you create a new data item without working with an <code>ArrayCollection</code> . An example of when this method is useful is a call center application where a customer fills in a form to create a single ticket item. In contrast, call center employees must see all the ticket items, so their application would fill an <code>ArrayCollection</code> with data items. Items are created automatically when you add them to an existing managed collection. When you call the <code>createItem()</code> method, you create a managed reference to that item that must be released explicitly. When you add an item to a managed collection, the reference to that item is released when you release the collection.

Method	Description
getItem()	Lets you get a single data item from the item identity. This method is useful when the application needs to get just a particular record and doesn't need require a <code>fill()</code> request to retrieve the entire collection. For example to renew a driver's license, you could call <code>getItem(ssn:xxx)</code> and <code>getItem(licenseNumber:xxx)</code> rather than <code>fill()</code> .
deleteItem()	Deletes an item which is managed using a <code>createItem()</code> , <code>getItem()</code> , or <code>fill()</code> method call. The delete is sent to the server as soon as the transaction is committed.
releaseItem()	Releases the specified item from management. If you hold onto <code>ItemReference</code> instances, you should call the <code>releaseItem()</code> method on the <code>ItemReference</code> instance to ensure that you release the proper reference when you might have made more than one call to the <code>getItem()</code> method to retrieve the same item from different parts of your client application. Calling the <code>releaseItem()</code> method releases any associated resources, including nested properties. The specified item no longer receives updates from the remote destination. In addition, if there are any uncommitted changes to this item and it does not appear in any other collection, the changes are also released. If the specified item exists in more than one collection, the value returned is a copy of the original unless the <code>allowCopy</code> parameter is set to <code>false</code> .

The following example shows a method that gets a specific data item when a DataGrid control changes. You get the item from the `ResultEvent.result` event. The identity value sent to the destination is `companyId`, which is set to the `companyId` of the currently selected item in the DataGrid control. The destination retrieves items based on their identity, which is specified in the destination definition in the `data-management-config.xml` configuration file.

```
<mx:Script>
    <! [CDATA[
    ...
        private function companyChange() {
            dsCompany.getItem({companyId: dg.selectedItem.companyId});
        }
    ]]>
</mx:Script>
...
```

Connecting to and disconnecting from a destination

A `DataService` component starts out disconnected from its server-side destination. The first time you perform an operation, the `DataService` component tries to connect to the destination. If the operation succeeds, a result event is sent to the `DataService` component. If the operation fails, a fault event is sent. You can call the `DataService.disconnect()` method to force a connected client to disconnect; in this case, the `DataService` component keeps a copy of its managed data and automatically resubscribes to pick up changes when you reconnect. You can also bind to the `DataService.connected` property, which indicates the current connected state of the `DataService` component.

You can also call `DataService.release()` method to release all managed objects fetched by a `DataService` component.

Working with managed objects on the client

There are two types of managed collections of objects: fills and managed associations. A fill is usually a query defined by a set of parameters called fill parameters, which should always identify a globally consistent set of values. Although one of the fill parameters can be a query string such as an HQL query for Hibernate, security can be compromised if you expose the ability for ad hoc queries to be formed from untrusted client code. A managed association is a relationship between two destination that you declare in a destination definition in the data-management-config.xml file; for more information, see “[Hierarchical data](#)” on page 293.

When you set the `autoSyncEnabled` property of the client-side DataService component to `true` before executing the fill, the results are synchronized with changes detected by other clients. In this case, the parameters of the fill method should identify a globally consistent set of values. In other words, if from some other session another client sends the same fill parameters, they get the same results.

Important: With the `autoSyncEnabled` property set to `true`, do not use session state to modify the results of the query without obscuring the fill parameters from other clients. You can use session state for security to allow the query or not. However, if you include data such as the user ID in the where clause of a query, you can expose that user data to another user that executes the same query. When the fill is refreshed, the same query is executed with the user session, which creates or updates an item or invokes the `refreshFill()` method of the server-side assembler. One way to avoid this problem entirely is by ensuring that two clients cannot use the same set of fill parameters. You include the user session ID or some other ID unknown to other clients as an additional fill parameter. Your assembler can ignore this fill parameter; it is used by the Data Management Service to keep the queries separate.

Typically, fill parameters are primitive values such as ints or strings that are used in queries. You can use complex objects, but in that case, the properties must serialize to the server and back again, and the values of the properties must be identical after they return. They should follow the value object pattern and override the `equals()` method. A copy of the fill parameters is sent to the server as part of the query and returned along with any message returned from the server that updates that collection, so avoid large object graphs as part of your fill parameters and ensure that they serialize to the server and back again.

Maintaining object identities

Each object managed by the Data Management Service must have one or more properties that uniquely identify the object value. These property values can be assigned on the server or on the client.

When the server assigns identities, the identity properties usually include the values of the primary key of the object. If you use identity properties that are not primitive types such as ints or Strings, make sure that your client-side and server-side identity properties serialize to each other as with other managed properties. In other words, they should follow the value object pattern. Identity properties that are not primitive types also must override the `java.lang.Object.equals()` and `java.lang.Object.hashCode()` methods. A FaultEvent is thrown when you specify a name that does not exist on the object in the `property` attribute of the `identity` element in a destination definition on the server.

When identities are assigned on the client, if you are using any syncing features, the identities must be unique across all clients. To avoid one client seeing the data of another client, you can include a value that is not known to other clients using the `mx.utils.UIDUtils.createUID()` method, or your session ID, or some other unique identifier you get from the server.

For more information, see “[Uniquely identifying data items](#)” on page 249.

About the DataStore object

The Data Management Service keeps outgoing and incoming data changes in a client-side object called DataStore. There are two modes for choosing the default DataStore instance: shared or not shared. If you use a shared DataStore instance, all DataService components that use the same channel set also use the same DataStore instance. A shared DataStore instance is used in the following situations:

- 1 The DataService component uses runtime configuration; it does not compile in the destination information using the `-services` compiler option.
- 2 The DataService component has one or more association element (for example, a `one-to-many` element).
- 3 The destination that a DataService component uses is referred to by an association element in another destination.

You can have independent DataStore instances if the data is totally independent. If you have an independent DataService component, the default depends on whether you specify your configuration in configuration files and compile the configuration in the SWF file. If you do compile in the configuration, you get an independent data store. If you do not, it isn't possible to discover whether you are using associations early enough in the initialization process, so a shared DataStore object is created.

If you want to change the default, you can create your own DataStore object and manually assign it to the `datastore` property of one or more DataService instances. Do this when you first initialize your DataService component before you have called any other DataService methods, such as `fill()` or `getItem()`.

Note: All DataService components that share the same DataStore must be using the same set of channels and the same setting for the `useTransactions` flag. Settings that are not consistent throw an error.

When you call a DataService component `commit()` method, it commits all of the changes for the DataStore object, which include the changes for all DataService components that share that DataStore object. Similarly, when you change one DataService component `autoCommit` property, the value changes for all DataService components that share the same DataStore object.

The `DataStore.CommitQueueMode` property controls how commits are queued to the server. By default, the client sends commit requests one at a time. In other words, the client waits for the result or failure of the first commit before sending the next one. If a batch fails, all messages are put back into the uncommitted queue so there is no possibility of receiving a batch with potentially stale changes. This is a conservative policy from a performance perspective. Many servers have conflict detection that would detect errors on those subsequent batches anyway, and can introduce unnecessary performance delays in certain data streaming applications.

You can also choose the `auto` mode. This mode checks for simple dependencies between any outstanding batches and the next batch to commit. If there are none, it sends the new batch. If there has been an update or delete request that depends on an outstanding create request, it holds the update or delete request until the create is completed. This imposes some overhead on each commit to check for dependencies. You can also set the value to `NOWAIT`. In this mode, the Data Management Service just commits the batches without checking dependencies. This is the fastest mode. However, there can be issues if the server does not properly handle update requests that depend on create requests without identity properties defined.

Handling errors

The DataService component dispatches fault events for errors that occur when processing an operation. This includes errors that occur when connecting to the server, and errors that an assembler class sends in response to processing an operation. For operations that you invoke in ActionScript, such as `fill()` and `commit()`, you can also listen for errors on that specific operation by adding a responder to the returned `AsyncToken`. For more information, see the documentation for the `mx.data.errors` package in the *Adobe LiveCycle ActionScript Reference*.

A DataService component automatically connects to a ChannelSet when it performs its first operation. When errors occur in your clients when processing a `commit()` request, the changes in the commit are put back into the uncommitted changes. You can choose to revert those changes by calling the DataService component `revertChanges()` method with an item argument, or you can call the `revertChanges()` method with no argument to revert all changes.

Routing data messages

Depending on your application requirements, you can use the auto synchronization feature to automatically route data messages between the server and client, or you can manually route data messages.

Automatically routing data messages

When you use the auto synchronization feature by setting a DataService component `autoSyncEnabled` property to `true` (default value), the server keeps track of every item and collection managed on each client. If you set the `cache-items` property to `true` (default value), it additionally caches a copy of all states managed by all active clients. If you set the `cache-items` property to `false`, it only stores the identity properties of all items managed by all active clients. The server also stores the list of client IDs subscribed to each item or collection. When a change is made to an item, the Data Management Service can quickly determine which clients have to be notified of that change even if the clients are managing randomly overlapping sets of data. The overhead for making this quick determination is potentially a large set of cached data on the server, and some overhead for maintaining the indexes and data structures required to do this notification as that data changes.

Manually routing data messages

Manually routing data messages is a more scalable way for Data Management Service clients to receive pushed updates. You can route data messages by manual routing specification rather than by using auto synchronization to automatically determine which clients get which objects.

Note: When you manually route messages, you cannot use the `autoRefresh fill` capability. Instead, if you want to update fills you must put in calls to manually manage fill membership.

For manual routing, you use the properties and methods of the `DataService.manualSync` property. The `manualSync` property is of type `mx.data.ManualSyncConfiguration`. This class lets you subscribe to changes made by other clients or on the server and it lets you control how your changes are published to other clients that subscribe to manually route data messages.

Using manual routing, you can route data updates to interested clients without requiring that the server maintain all the in-memory sequence and client data structures necessary to support the auto-sync-enabled case. Clients that participate in manual routing must explicitly subscribe to topics or use selector expressions that describe the changes they want to receive. Similarly, they also specify topics or metadata that are attached to changes they make. The server then routes the changes of a particular client to any clients subscribed to matching criteria. This system is implemented using the `MultiTopicConsumer` and `MultiTopicProducer` classes, but the client interacts with them through the `manualSync` property of the DataService component.

The `manualSync` property of the DataService component contains several types of properties: those that affect message producer behavior and those that affect the message consumer behavior. The producer properties let you attach metadata to the changes produced by a client (subtopics and default headers). The consumer properties and methods control whether a client is subscribed to changes from the server and, if subscribed, whether it provides a list of the subscriptions the client has. If the metadata that a producer attaches to a change matches any one subscription of the consumer, that consumer receives the message.

For changes made by using the server push API (DataServiceTransaction), you can specify the producer properties on DataServiceTransaction. Clients can subscribe to receive changes that match a given pattern. Changes that match that pattern can be produced on another client or on the server push API.

By using subtopics, you can easily and efficiently form separate isolated groups (for example, meetings or other collaborative groups). These groups can overlap through the use of multiple subtopics in either the producer or consumers. Also, a producing client does not have to consume the messages it produces, which provides additional flexibility.

For each subscription, you can also specify a selector expression in the Consumer component `selector` property to select messages the client wants to receive. The subtopics can be used to do OR functionality. Selector expressions provide a more flexible way to do intersected groups (AND and value-based conditionals). On a Producer, you can set the `defaultHeaders` property to evaluate the selector expression for a given Consumer component. Selector expressions are less efficient to implement on the server. Each unique selector expression must be evaluated for each message sent to that destination.

To use manual routing, you must set the `allow-subtopics` element in the destination definition in the `data-management-config.xml` file to `true`, as the following example shows:

```
<destination id="inventory">
    <adapter ref="java-dao" />
    <properties>
        <source>flex.samples.product.ProductAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="productId"/>
        </metadata>
        <server>
            <allow-subtopics>true</allow-subtopics>
        </server>
    </properties>
</destination>
```

Using the `manualSync` property

You use the `DataService.manualSync` property to perform manual routing on a client. Each client independently defines its `manualSync` configuration. The `manualSync` property is of type `ManualSyncConfiguration`, and you can use the following properties and methods directly on the `manualSync` property:

Properties and methods	Description
<code>producerSubtopics</code>	The list of subtopics each change is published to.
<code>producerDefaultHeaders</code>	An Object containing name-value pairs that is passed along with outbound data messages from the client. The server evaluates these values against the selector expressions defined by other clients to determine which clients to push the results to.
<code>consumerSubscriptions</code>	This is a list of <code>SubscriptionInfo</code> instances. Each <code>SubscriptionInfo</code> has a <code>subtopic</code> and a <code>selector</code> expression.
<code>consumerSubscribe(clientId)</code> , <code>consumerUnsubscribe()</code>	Subscribing for changes on the server is a two-step process. First you add subscriptions, and then you call <code>consumerSubscribe()</code> .
<code>consumerAddSubscription(subtopic, selector)</code> , <code>consumerRemoveSubscription(subtopic, selector)</code>	Convenience methods to update the <code>consumerSubscriptions</code> property of the <code>ManualSyncConfiguration</code> .

The following example shows ActionScript code for establishing manual routing on a client:

```
...
<mx:Script>
<! [CDATA[
...
var ds:DataService = new DataService("Meeting");
ds.autoSyncEnabled = false;
ds.manualSync.producerSubtopics.addItem("flex-room");
ds.manualSync.consumerAddSubscription("flex-room");
ds.manualSync.consumerSubscribe();
ds.fill(...);
...
]]>
</mx:Script>
```

Using manual routing in a clustered environment

If a client uses `autoSyncEnabled`, the Data Management Service broadcasts changes to all cluster members. The Data Management Service is not globally aware of whether any clients are using `autoSyncEnabled`. To improve scalability when manually routing messages, you can set the `use-cluster-broadcast` element in the `server` element of the Data Management Service to `false`. By default, you configure the Data Management Service in the `data-management-config.xml` file.

The following example shows a `use-cluster-broadcast` element set to `false`:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="data-service" class="flex.data.DataService">
...
<properties>
    <use-cluster-broadcast>false</use-cluster-broadcast>
</properties>
</service>
```

Controlling whether clients receive pushed changes

By default, the Data Management Service detects data changes made from Flex clients and the server push API and propagates those changes to other clients. You can change this default behavior for a specific destination by setting `auto-sync-enabled` to `false` in the destination configuration.

To turn on the auto-sync behavior for a specific ArrayCollection on the client side, set the DataService component `autoSyncEnabled` property to `true` before you call its `fill()` method. Similarly, to retrieve a managed item from an individual reference, you set the DataService component `autoSyncEnabled` property to `true` before you call its `getItem()` or `createItem()` method.

Changing the value of the `autoSyncEnabled` property does not affect existing managed objects on that client. It only affects `fill()`, `getItem()` and `createItem()` method calls that you make after changing the value. This lets you have managed instances with the `autoSyncEnabled` property set to `true` and others with the `autoSyncEnabled` property set to `false` from the same destination in the same client.

Controlling pushed changes

By default, when a client detects changes made on the server or by other clients, the changes are immediately applied to the client. You can turn this functionality off for a given destination by setting a DataService component `autoMerge` property to `false`. When pending changes come in, the `mergeRequired` property is set to `true` on the `DataStore` object. To merge the pending changes, you call the `DataService.merge()` method. To avoid conflicts, merge changes before you modify any data locally on the client.

Note: A client does not make page requests while unmerged updates exist if a DataService component autoMerge property is set to false, paging is enabled, and the page size is a nonzero value. You can test for this case by checking the value of the mergeRequired property of the DataService component or its associated DataStore object.

Releasing paged items from a collection

When paging is enabled, you can remove pages of data sent from the server to the client to optimize performance when dealing with large data sets. To remove pages of data from an ArrayCollection object, you use the DataService releaseItemsFromCollection(), isRangeResident(), and isCollectionPaged() methods.

The releaseItemsFromCollection() method releases a range of items in the ArrayCollection. When paging through a large collection, you might want to free resources occupied by items in the collection and stop subscribing for updates for those items. You specify the startIndex of the first item that you want to release and the number of items to release. If an item at that position is not yet paged, that index is skipped.

The isRangeResident() method returns true if the supplied range of items is all paged in. The isCollectionPaged() method returns true if the passed collection is using paging features. If the collection is paged, it is safe to pass the collection to the releaseItemsFromCollection() method to release individual items in the collection from management.

The following example shows a method in an MXML application that uses the isCollectionPaged() and releaseItemsFromCollection() methods to determine if data in a DataGrid control is paged and then release items that are not in the displayed range as you scroll through the grid:

```
...
private function doUpdateManagedScrollGrid(scroll:Object, dataService:DataService,
    collection>ListCollectionView, visibleRowCount:int):void {
    trace("scroll position: " + scroll.event.position + " row count: " + visibleRowCount +
        " length: " + collection.length);
    //scrollCurrentPosition = scroll.event.position;
    //scrollVisibleMax = visibleRowCount + scrollCurrentPosition;
    // Nothing to optimize here if paging is not enabled.
    if (!dataService.isCollectionPaged(collection))
        return;

    // The first index we want to fetch
    var top:int = Math.min(Math.max(scroll.event.position - PAD - 1, 0),
        collection.length);

    // round up to the start of the last page
    top = top - (top % PAGE_SIZE);

    // The last index we want to fetch
    var bot:int = Math.min(scroll.event.position + visibleRowCount + PAD,
        collection.length);
```

```
// round down to the end of the next page size
bot = bot + (PAGE_SIZE - (bot % PAGE_SIZE));
var ct:int;
ct = dataService.releaseItemsFromCollection(collection, 0, top);
trace("released: " + ct + " from: 0-" + top);
if (bot < collection.length)
    ct = dataService.releaseItemsFromCollection(collection, bot,
        Math.max(collection.length- bot, 0));
trace("released: " + ct + " from: " + bot + "-" + (collection.length - bot));

try {
    trace("fetching: " + top + ":" + (bot-top));
    collection.getItemAt(top, bot-top);
    trace("all resident");
}
catch (ipe:ItemPendingError) {
    trace("pending...");
}
}

...
...
```

Mapping client-side objects to Java objects

To represent a server-side Java object in a client application, you use the `[RemoteClass(alias=" ")]` metadata tag to create a strongly typed ActionScript object that maps directly to the Java object. You specify the fully qualified class name of the Java class as the value of `alias`. This is the same technique that you use to map to Java objects when using `RemoteObject` components.

You can use the `[RemoteClass]` metadata tag without an alias if you do not map to a Java object on the server but you do send back your object type from the server. The ActionScript object is serialized to a `Map` object when it is sent to the server, but the object returned from the server to the clients is your original ActionScript type. Both the client-side and server-side classes must contain an empty constructor. Also, the compiled SWF file must contain a reference to your ActionScript class. If the SWF file does not contain a reference to that class, a generic `ASObject` instance is created instead of the desired object type.

To create a managed association between client-side and server-side objects, you also use the `[Managed]` metadata tag or explicitly implement the `mx.data.IManaged` interface.

The CRM application that is included in the Adobe LiveCycle Data Services sample applications provides a good example of a managed association. The following example shows the source code for the client-side ActionScript `Company` class, which has a managed association with the server-side Java `Company` class:

```
package samples.crm
{
    [Managed]
    [RemoteClass(alias="samples.crm.Company")]
    public class Company
    {
        public var companyId:int;
        public var name:String = "";
        public var address:String = "";
        public var city:String = "";
        public var state:String = "";
        public var zip:String = "";
        public var industry:String = "";
        public function Company()
        {
        }
    }
}
```

The following example shows the source code for the corresponding server-side Java Company class. This example shows properties defined using the JavaBean style getX/setX syntax. You also can define public fields just as you do in ActionScript.

```
package samples.crm;

import java.util.Set;

public class Company
{
    private int companyId;
    private String name;
    private String address;
    private String city;
    private String zip;
    private String state;
    private String industry;

    public String getAddress()
    {
        return address;
    }

    public void setAddress(String address)
    {
        this.address = address;
    }

    public String getCity()
    {
        return city;
    }

    public void setCity(String city)
    {
        this.city = city;
    }
}
```

```
public int getCompanyId()
{
    return companyId;
}

public void setCompanyId(int companyId)
{
    this.companyId = companyId;
}

public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}

public String getState()
{
    return state;
}

public void setState(String state)
{
    this.state = state;
}

public String getZip()
{
    return zip;
}

public void setZip(String zip)
{
```

```
        this.zip = zip;
    }

    public String getIndustry()
    {
        return this.industry;
    }

    public void setIndustry(String industry)
    {
        this.industry = industry;
    }

    public String toString()
    {
        return "Company(companyId=" + companyId + ", name=" + name + ",
               address=" + address +
               ", state" + state + ", zip=" + zip + " industry=" + industry + ")";
    }
}
```

Define a computed property in ActionScript

There can be times when you want your client-side managed classes to expose properties that are computed from the value of other properties. These types of properties tend to be read-only, but you want them to update properly when either of the source properties changes. It is fairly easy to add these constructs to your client-side managed classes. If you make these properties part of your data model, it is easier to show them as a column in a DataGrid component or use them to display a row in a list from an ItemRenderer object. Make sure you define these properties as transient with the `[transient]` metadata tag, since you do not want data management to track changes made to them or to serialize them to the server. You also must send `PropertyChange` events whenever either of the source properties has changed. The following code snippet implements a read-only `displayName` property by concatenating the `firstName` and `lastName` properties:

```
private var firstName:String;
private var lastName:String;
[Transient]
public function get displayName():String {
    return _firstName + " " + _lastName;
}
public function get firstName():String {
    return _firstName;
}
public function set firstName(fn:String):void {
    var oldDN:String = displayName;
    firstName = fn;
    var newDN:String = displayName;
    dispatchEvent(PropertyChangeEvent.createUpdateEvent
        (this, "displayName", oldDN, newDN));
}
public function get lastName():String {
    return _lastName;
}
public function set lastName(ln:String):void {
    var oldDN:String = displayName;
    lastName = ln;
    var newDN:String = displayName;
    dispatchEvent(PropertyChangeEvent.createUpdateEvent
        (this, "displayName", oldDN, newDN));
}
```

Handling data synchronization conflicts

When working with distributed data applications, there are often times when clients try to make data changes that conflict with changes that were already made. The following scenarios are core cases where data synchronization conflicts occur:

Conflict type	Description
Update with stale data	The server detects a conflict when a client commits changes because the data has changed since the client received the data.
Client is pushed changes when it has uncommitted, conflicting changes	A client is in the midst of changing the data that conflicts with changes pushed to it from the server.
Delete with stale data	The server detects a conflict when the client tries to delete an item with a stale version of that object.
Client is pushed changes to an item it has deleted locally	A client has an uncommitted request to delete data when an update has just occurred to the same data, and a data message has been pushed to the client from the server.

Conflict type	Description
Update to deleted data without client push	A client tries to update data that has already been deleted from the server.
Update to deleted date with client push	A client tries to update data that has already been deleted, and a data message has been pushed to the client from the server. The client tries to send the update, but there is a conflict on the client.
Client sends update and receives REMOVE_ITEM_FROM_FILL message for same item.	A client updates an item and then receives a REMOVE_ITEM_FROM_FILL message from the server for that item. You can determine whether the update/item-gone-from-the-server conflict is a delete versus a remove-from-fill conflict by testing the <code>serverObjectDeleted</code> flag on the Conflict object. The flag is <code>true</code> when the conflict is not caused by an item being removed on the server (<code>serverObject</code> is not <code>null</code>). It is <code>false</code> when the conflict is caused by a remove-item-from-fill on the server (<code>serverObject</code> is <code>null</code>).

The following example shows a simple event handler that displays an Alert box on the client where a conflict occurs, and accepts the server version of the data:

```
...
public function useDS:void {
    ...
    ds.addEventListener(DataConflictEvent.CONFLICT, conflictHandler);
}

public function conflictHandler(event:DataConflictEvent):void {
    var conflicts:Conflicts = ds.conflicts;
    var c:Conflict;
    for (var i:int=0; i<conflicts.length; i++) {
        c = Conflict(conflicts.getItemAt(i));
        Alert.show("Reverting to server value", "Conflict");
        c.acceptServer();
    }
}
```

You can call `Conflicts.getConflict(Item)` to test if there is a conflict for a specific item.

For more information about the `mx.data.Conflicts` and `mx.data.Conflict`, and `mx.data.events.DataConflictEvent` classes, see the *Adobe LiveCycle ActionScript Reference*.

When you use the Java adapter with a custom assembler, you can write logic in your assembler class that handles data synchronization and throws a `DataSyncException` if the previous version of the data does not match what is currently in the data resource. A `DataSyncException` results in a `DataConflictEvent` on the client. For more information, see “Custom assemblers” on page 254.

You can also let the user decide which version of data gets used when a conflict occurs, as the following code snippets show. `PersonForm` represents a custom component that displays all of the properties of the `Person` object assigned to its `dataSource` property.

```
// Conflict event handler:  
function resolveConflictsHandler():void {  
    displayConflictsScreen();  
}  
...  
<!-- Conflicts screen MXML code: -->  
<PersonForm id="serverValue" editable="false" dataSource="{ds.conflicts.current.serverObject}" />  
  
<PersonForm id="clientValue" dataSource="{ds.conflicts.current.clientObject}" />  
  
<PersonForm id="originalValue" dataSource="{ds.conflicts.current.originalObject}"  
editable="false"/>  
  
<mx:Button label="Accept Server" click="ds.conflicts.current.acceptServer()" />  
<mx:Button label="Accept Client" click="ds.conflicts.current.acceptClient()" />  
...
```

Data Management Service configuration

To enable distributed data in Flex client applications, you connect to a server-side Data Management Service destination. You configure destinations as part of the Data Management Service definition in a configuration file.

For information about using the client-side DataService component and connecting to a Data Management Service destination in MXML or ActionScript, see “[Data Management Service clients](#)” on page 226.

Note: With the modeling features introduced in LiveCycle Data Services 3, you can use model-driven development to take advantage of advanced Data Management Service features without writing Java code or configuring destinations on the server. This content contains information about configuration for non-model development and model-driven development. Model annotations for model-driven development are described in Application Modeling with Adobe LiveCycle® Data Services. For additional information, see “[Model-driven applications](#)” on page 326.

About Data Management Service configuration

The most common configuration tasks for the Data Management Service are defining destinations, applying security to destinations, and modifying logging settings. For information about security and logging, see “[Security](#)” on page 429 and “[Logging](#)” on page 420.

You configure Data Management Service destinations in the data-management-config.xml file. You can also dynamically configure services by using the run-time configuration feature; for more information, see “[Run-time configuration](#)” on page 411.

A Data Management Service destination uses an adapter that provides the infrastructure for interacting with data resources. The two adapters included in LiveCycle Data Services are the ActionScript Object Adapter and the Java Adapter.

The properties available for a destination depend on the type of adapter you use. For example, when you use the Java Adapter, you can configure source, scope, and other properties.

You specify data adapters in a destination. You can also specify adapter-specific settings. For general information about adapters, see “[Key concepts of data management](#)” on page 222.

Understanding data management adapters

An *adapter* is responsible for updating the persistent data store, or in-memory data in the case of the ActionScript Object Adapter, in a manner appropriate to the specific data store type. The Java Adapter delegates this responsibility to the assembler class.

The Java Adapter and the ActionScript Object Adapter are included with Adobe LiveCycle Data Services. The Java Adapter passes data changes to methods available on a Java class, called a Java assembler, which handles interaction with the underlying data resource.

You use the ActionScript Object Adapter when a Flex client application is the only creator and consumer of transient data objects and there is no back-end data resource. The Data Management Service uses the ActionScript Object Adapter to manage objects in the server's memory.

The Java Adapter lets you synchronize Java objects on the server with ActionScript objects on the client. This adapter passes data changes to methods available in a Java class called an assembler. The following table describes the types of Java assemblers:

Assembler	Description
Custom assembler	Use a custom assembler if you have sophisticated data models and understand Java server-side code. This type of assembler offers the most flexibility, but is also the most complex to use. For more information, see “ Custom assemblers ” on page 254.
SQL Assembler	The SQL Assembler is a specialized assembler that provides a bridge from the Data Management Service to a SQL database management system. Use this type of assembler if your database data model does not have complex object relationships and you want to expose that data model to MXML without writing Java server-side code. For more information, see “ Standard assemblers ” on page 268.
Hibernate Assembler	The Hibernate Assembler is a specialized assembler that provides a bridge from the Data Management Service to the Hibernate object-relational mapping system. Use this assembler if you do not want to write a custom assembler, you require access to database objects, you are using or have used the Hibernate object-relational mapping system, and you are comfortable with Java. For more information, see “ Standard assemblers ” on page 268.
Model Assembler	You use the Model Assembler for model-driven development. LiveCycle Data Services generates Model Assembler instances based on entities defined in a model. For more information, see “ Model-driven applications ” on page 326.

When using the ActionScript Object Adapter, you can use any field in an ActionScript object to provide a unique identity property in the `identity` element of the destination's `metadata` section. You can establish a composite identity by specifying more than one identity property. If unspecified, by default the `uid` property of the ActionScript object is the identity property.

The following destination definition, which you create in the `data-management-config.xml` file, specifies a single identity property. For more information about destinations, see “[About Data Management Service destinations](#)” on page 246.

```
<destination id="notes">
    <adapter ref="actionscript"/>
    <properties>
        <metadata>
            <identity property="noteId"/>
        </metadata>
    </properties>
</destination>
```

Each item must have a unique identity property and there is no back-end data resource that automatically creates identity properties. Therefore, for an application in which you require an ArrayCollection of data items, you must generate item identity properties on the client when you create managed objects. The code in the following example uses the `mx.utils.UIDUtil.createUID()` method to generate a universally unique `customer.custId` property to serve as the identity property. You could also specify `custId` as the identity property in a destination definition in the `data-management-config.xml` file.

```
private function newCustomer():void {
    dg.selectedIndex = -1;
    customer = new Customer();
    customer.custId = mx.utils.UIDUtil.createUID();
}
```

The following example shows a client application that uses the ActionScript Object Adapter to persist a single data item if it doesn't already exist in a TextArea control across all clients. When the server is stopped, the data is lost because there is no back-end data resource.

```
<?xml version="1.0"?>
<! -- ds\datamanagement\ASAdapter.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
height="100%" width="100%"
creationComplete="initApp() ; ">

<mx:Script>
<! [CDATA[
import mx.data.DataService;
import mx.rpc.AsyncToken;

public var noteObj:Object = new Object();
[Bindable]
public var getToken:AsyncToken;
private var ds:DataService;

private function initApp():void {
ds = new DataService("notes");
ds.autoCommit = false;
noteObj.noteId = 1;
noteObj.noteText =
"Type your notes here and share them with other clients!";
// The first getItem method parameter is the identity property of noteObj.
getToken = ds.getItem({noteId:noteObj.noteId}, noteObj);

}
]]>
</mx:Script>
<mx:Binding source="log.text" destination="getToken.result.noteText"/>
<mx:TextArea id="log" width="100%" height="100%" text="{getToken.result.noteText}"/>
<mx:Button label="Send" click="ds.commit();"/>
</mx:Application>
```

About Data Management Service destinations

A *Data Management Service destination* is the endpoint that you send data to and receive data from when you use the Data Management Service to provide data distribution and synchronization in your applications.

Note: For model-driven development with the Model Assembler, you do not configure destinations in configuration files. LiveCycle Data Services configures Model Assembler destinations automatically when a model is deployed to the server.

You configure Data Management Service destinations in the data-management-config.xml file. By default, the data-management-config.xml file is located in the WEB_INF/flex directory of the web application that contains Adobe LiveCycle Data Services. The data-management-config.xml file is referenced in the top-level services-config.xml file. You can also dynamically configure services by using the run-time configuration feature; for more information, see “Run-time configuration” on page 411.

The following example shows a basic Data Management Service configuration. It contains one destination that uses the Java Adapter to interact with a data resource.

```
<destination id="inventory">
    <adapter ref="java-dao" />
    <properties>
        <source>flex.samples.product.ProductAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="productId" />
        </metadata>

        <network>
            <paging enabled="false" pageSize="10" />
        </network>
    </properties>
</destination>
```

A Data Management Service destination references one or more channels that transport messages, and contains network- and server-related properties. It can also reference a *data adapter*, which is server-side code that lets the destination work with data through a particular type of interface, such as a Java object. You can also configure a specific data adapter as the default data adapter, in which case you do not have to reference it inside the destination. A destination can also reference or define security constraints for a destination.

To use the Java Adapter, you write an assembler class or use the HibernateAssembler or SQLAssembler class. An *assembler class* is a Java class that interacts indirectly or directly with a data resource. When you use either the HibernateAssembler class or the SQLAssembler class, you typically start with the existing Assembler class, and only have to write Java code if you extend the functionality they provide. A common design pattern is for the assembler to call a data access object (DAO) that calls a data resource.

General Data Management Service configuration

Some aspects of configuring Data Management Service destinations apply to most destinations, and others are determined by the particular data adapter that a destination uses. This section describes general configuration for Data Management Service destinations.

Setting the scope of an assembler

Use the `scope` element of a Data Management Service destination to specify whether the assembler is available in the request scope (stateless), the application scope, or the session scope.

Note: When you use model-driven development with the Model Assembler, assembler instances are always in the application scope.

When a client invokes a `fill()`, `update()`, `delete()`, or `create()` method, a request (DataMessage) is sent to the server destination for processing. The destination delegates to its configured assembler to perform the processing.

When you set scope to request, a new assembler instance is created to process each and every inbound request.

When you set scope to session, assembler instances are created and then cached individually for each client session. You would use a session-scoped assembler when you want to code your assembler to maintain some internal state for the client across calls, or you want to access the assembler instance for a given client from somewhere else (for example, while handling a remoting request), by looking up the assembler instance in the session. When using a session-scoped assembler, it is essential to properly synchronize changes to any internal state (variables/properties) that the assembler defines. When you set scope to application, a single assembler instance is created and cached at an application-wide level. That single assembler instance handles all requests from all clients that target the destination. In this scenario, you generally want to avoid any mutable state in the assembler; many clients can concurrently use a stateless assembler without issue. For most applications, your state is stored and managed in a shared transactional backend, such as a database, so a stateless assembler works well. If you do want to track any state in an application-scoped assembler, you must protect reads and writes of that state with Java synchronization (synchronized keyword or explicit lock objects). In almost all cases, a Data Service destination is best served by an application-scoped assembler that exposes a portion of a transactional shared backend, which is generally a database. Scenarios for a request-scoped assembler are few and far between. A session-scoped assembler is generally only useful if you want to access the assembler instance from other code executing on the server by looking up the assembler instance in the session.

Configuring paging

LiveCycle Data Services gives several options for paging data from the server to the client. For more information, see “[Data paging](#)” on page 303.

Using transactions

By default, LiveCycle Data Services encapsulates client-committed operations in a single J2EE distributed transaction. All client data changes are handled as a unit. If one value cannot be changed, changes to all other values are rolled back. Distributed transactions are supported as long as your Java 2 Enterprise Edition (J2EE) server provides support for J2EE transactions (JTA).

When you are not using model-driven development, you enable or disable transactions in the data-management-config.xml file. The boldface text in the following example shows the `use-transactions` property set to `true` (the default value) at the service level so that it applies to all Data Management Service destinations:

```
...
<service id="data-service" class="flex.data.DataService">
...
    <properties>
        <b><use-transactions>true</use-transactions></b>
    </properties>
</service>
...
```

If you set the `use-transactions` property to `false`, each assembler method can create and commit its own transaction. The batch of changes processes until the first error occurs. It is assumed that all changes that precede the error are applied to the data store. When these changes make it back to the client, their result handler is triggered. The change that caused the error is reported as a fault on the client. All changes that occur in the batch after the error are not sent to the assembler and are also reported as faults on the client.

For model-driven development, you enable or disable transactions in the `use-transactions` annotation in an entity, as the following example shows:

```
<entity name="Book" persistent="true">
    <annotation name="DMS">
        <item name="use-transactions">
            true
        </item>
    </annotation>
    ...
</entity>
```

Uniquely identifying data items

A Data Management Service destination contains one or more identity properties that you can use to designate data properties to be used to guarantee unique identity among items in a collection of objects. In the case of the SQL assembler, an identity property maps to a database field.

The boldface text in the following example shows an identity property in a destination:

```
...
<destination id="contact">
    <properties>
    ...
        <metadata>
            <b><identity property="name"/></b>
        </metadata>
    </properties>
</destination>
...
```

For model-driven development, you configure identity properties in id child elements of a persistent entity, as the boldface text in the following example shows. The type attribute specifies the data type of the identity property. Model-driven service providers and code generators are only required to support integer.

```
<entity name="QuoteRequest" persistent="true">
    <id name="id" type="integer">
    <property name="state" type="State"/>
    <property name="carModel" type="CarModel" />
    <property name="suspensionDetails" type="SuspensionDetail[]" />
</entity>
```

When you are not using model-driven development, the identity element takes an optional type attribute, which is the Java class of the specified identity property. You must use this when your identity type may not have a one-to-one correspondence with ActionScript types. The most common problem is that an ActionScript Number is converted to either a long or an integer, depending on its value. The code in the following example shows an identity element with a type attribute:

```
<identity property="id" type="java.lang.Long">
```

A common mistake is to try to use one identity element for all properties, as the following example shows:

```
<metadata>
    <!-- Don't do this. -->
    <identity property="firstName,lastName"/>
</metadata>
```

Instead, you can specify multiple identity elements for a multiple field identity, as the following example shows:

```
<metadata>
    <identity property="firstName"/>
    <identity property="lastName"/>
</metadata>
```

The `identity` element can optionally take an `undefined-value` attribute, which you use to treat the identity of a property (usually of type `int`) as undefined. When an object is initially created on the client, often its identity is not known until the item is created on the server. In other situations, the client code defines the identity property value before the item is created by the Data Management Service. If `null` is not valid for your data type, for example, if you have an `int` or `Number` as your identity property, you set the `undefined-value` attribute on the `identity` element to specify the undefined value. Often the number 0 (zero) is not a valid identity, so this would be the undefined value, as the following example shows:

```
<metadata>
  <identity property="id" undefined-value="0"/>
</metadata>
```

If you do not specify an undefined value, the Data Management Service does not let you create two objects at the same time because both have the same identity value.

Using identity to manage relationships between objects

The Java Adapter supports flat objects that only contain primitive fields and properties, as well as objects that have complex objects as properties. If you have complex objects as properties and those objects have their own identity property or properties, you can choose to define a separate destination to manage the synchronization of those instances based on the identity property or properties specified in an `identity` element in the destination definition. The following example shows a destination named `contact` that has a many-to-one relationship with a destination named `account`. The `account` destination has a complex property named `contact` and the `contact` destination manages each contact instance.

```
<!-- Child destination -->
<destination id="contact">
  <properties>
    <metadata>
      <identity property="contactId" undefined-value="0"/>
      <!-- Reference to account destination that has a "contact" property -->
      <many-to-one property="account" destination="account" lazy="true"/>
    </metadata>...
  </properties>
</destination>
```

If there is no `identity` property for complex object properties and each instance is owned by its parent object, you can use the parent destination to manage the state of the object properties. For more information, see “[Hierarchical data](#)” on page 293.

Caching data items

By default, the Data Management Service caches items returned from `fill()` and `getItem()` calls and uses cached items to implement paging and to build object graphs on the server when implementing lazy associations. This causes a complete copy of the managed state of all active clients to be kept in each server's memory.

When you are not using model-driven development, you can turn off item caching by setting `cache-items` to `false`, as the following example shows:

```
...
<destination id="contact">
  <properties>
...
    <cache-items>false</cache-items>
...
  </properties>
</destination>
...
```

If you set `cache-items` to `false`, you must specify a method to support paging or lazy associations in a `get-method` element when using this destination. When the `cache-items` property is `false`, the Data Management Service only caches the identity properties of the items on the server. This greatly reduces the footprint of data kept on the server, but there is still some memory used on the server for each managed object on each client. To eliminate this overhead entirely, you set `DataService.autoSyncEnabled` to `false` on the client, and either manually refresh clients or use the manual synchronization feature to route changes to clients. For information about manual synchronization, see “[Manually routing data messages](#)” on page 234.

For model-driven development, you disable item caching by setting the `cache-items` annotation of an entity to `false`, as the following example shows:

```
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="cache-items">false</item>
  </annotation>
  ...
</entity>
```

Synchronizing data automatically

When you are not using model-driven development, the `auto-sync-enabled` element controls the default value of the client-side `DataService.autoSyncEnabled` property for clients that are using a destination. The default value of this element is `true`.

The following example shows an `auto-sync-enabled` element:

```
...
  <destination id="contact">
    <properties>
      ...
        <auto-sync-enabled>false</auto-sync-enabled>
      ...
    </properties>
  </destination>
  ...

```

For model-driven development, you set auto synchronization in the `auto-sync` annotation of an entity, as the following example shows. The default value is `true`.

```
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="auto-sync-enabled">
      false
    </item>
  </annotation>
  ...
</entity>
```

Using strong and anonymous types

Note: This topic does not apply to model-driven development with the Model Assembler.

The properties of a strongly typed object are defined at compile time by using declared fields or get and set methods. An anonymous type is usually represented in Java by a `java.util.Map`, and in ActionScript by an instance of type `Object`. For anonymous objects, type checking is performed at run time, not compile time.

You can use either strongly typed objects or anonymous objects with the Data Management Service on either the client or the server. For example, to use anonymous objects on both the client and server, your assembler returns `java.util.Map` instances that are converted to `Object` instances on the client. When the client modifies those `Object` instances, they are sent back to the assembler as `java.util.Map` instances.

To use strongly typed objects on the client and server, you define an ActionScript class that has explicit public properties for the data you want to manage with the `[Managed]` metadata tag. You map this class to a Java class on the server by using the `[RemoteClass(alias="java-class-name")]` metadata tag. Your Data Management Service destination in this case does not have to be configured with either class explicitly, which allows a single destination to support whatever instances your client sends to it. The client just has to expect the instances returned by the Java assembler, and the Java assembler has to recognize instances returned by the client. This allows one destination to support an entire class hierarchy of instances and you do not have to configure the Data Management Service destination explicitly for each class as long as they all have the same identity and association properties.

Using item-class to convert to anonymous ActionScript objects

The Data Management Service supports a technique that allows your Java assembler to return strongly typed Java instances that are converted to anonymous types on the client. In this case, you do not have to maintain ActionScript classes for each of your Java classes. When those anonymous instances are modified, the server receives them as `java.util.Map` instances. Data Management Service supports the `item-class` element in the `properties` section of a destination definition to automatically convert these instances back into a single strongly typed Java class that the Java assembler expects. Set the `item-class` element to refer to the Java class that your assembler expects. This pattern supports only the common case where each destination returns only instances of a single Java class. If your assembler must operate on a class hierarchy and you do not want to use strongly typed instances on the client, you must convert from the `java.util.Map` to your strongly typed instances in your assembler yourself.

You can also write a custom assembler that uses anonymous objects on the server that are serialized to strongly typed ActionScript classes on the client. To do this, you create an instance of the class `flex.messaging.io.amf.ASObject` that implements the `java.util.Map` interface. Set the `type` property of that instance to be the fully qualified class name of the ActionScript class that you want that instance to serialize to on the client, and store the property values that the ActionScript instance expects as key-value pairs in the `java.util.Map`. When that object is received by the client, the serialization code creates an instance of the proper type. Ensure that you declare an instance of the ActionScript class in your client-side code even if you are not going to use it. You must have a reference to the class to create a dependency on it so that it stays in the SWF file, as the following example shows:

```
public var myclass:MyClass;
```

Using operation-specific security constraints

For destinations that use the ActionScript Object Adapter or the Java Adapter, you can reference security constraints in the following elements, which are child element of the `server` element in a destination. These elements apply security constraints to the corresponding create, read, update, delete, and count operations.

Element	Description
<code><create-security-constraint ref="sample-users"/></code>	Applies the security constraint referenced in the <code>ref</code> attribute to the create requests.
<code><read-security-constraint ref="sample-users"/></code>	Applies the security constraint referenced in the <code>ref</code> attribute to fill requests.

Element	Description
<update-security-constraint ref="sample-users"/>	Applies the security constraint referenced in the <code>ref</code> attribute to update requests.
<delete-security-constraint ref="sample-users"/>	Applies the security constraint referenced in the <code>ref</code> attribute to delete requests.
<count-security-constraint ref="sample-users"/>	Applies the security constraint referenced in the <code>ref</code> attribute to count requests.

For model-driven development, you apply operation-level security constraints in `operationname-security-constraint-ref` annotations in an entity, as the following example shows:

```
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name=
      "update-security-constraint-ref">
      sample-users
    </item>
  </annotation>
  ...
</entity>
```

Setting a reconnection policy

Use the optional `reconnectFetch` property to determine what the client-side DataService component fetches in the event of a reconnection to the remote destination. There are two options: `IDENTITY` (just retrieve the sequence id) and `INSTANCE` (retrieve the complete contents of the fill). The default value is `IDENTITY`.

The following example shows a `reconnectFetch` property set in a destination:

```
...
  <destination id="contact">
    <properties>
    ...
      <network>
      ...
        <reconnect fetch=" INSTANCE" />
      </network>
    ...
    </properties>
  </destination>
```

For model-driven development, you set a reconnection policy in a `reconnectFetch` annotation in an entity, as the following example shows:

```
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="reconnect-fetch">
      INSTANCE
    </item>
  </annotation>
  ...
</entity>
```

Custom assemblers

If you are using complex data models and understand Java server-side code, you can create custom assemblers to use with the Java Adapter. This type of assembler offers the most flexibility for handling specific application requirements, but requires you to write more code than the standard assemblers.

Using custom assemblers

Depending on the application objects you want the Data Management Service to manage, you define one or more destinations in the data-management-config.xml file. Each destination maps directly to an application object that you want to manage. You define an assembler class for each destination.

Your assembler class only has to support the methods that correspond to the operations that the clients use. Your assembler usually implements at least one fill method. Fill methods implement queries, each of which returns a collection of objects to the client code. If your client must update objects, your assembler must include a sync method or individual create, update, and delete methods.

Depending on your server architecture, you directly transfer your domain model objects to the client or implement a distinct layer of data transfer objects (DAOs) that replicate only the model state you want to expose to your client tier.

The Data Management Service supports two approaches to writing custom assembler classes: the Assembler interface approach and the fill-method approach. The Assembler interface approach is a more traditional way to build Java components. You extend the `flex.data.assemblers.AbstractAssembler` class, which implements the `flex.data.assemblersAssembler` interface. You override methods such as the `getItem()`, `fill()`, `createItem()`, and `updateItem()` methods. The `AbstractAssembler` class provides default implementations of all of the Assembler interface methods. The `AbstractAssembler` class and Assembler interface are documented in the LiveCycle Data Services Javadoc API documentation. For more information, see “[The Assembler interface approach](#)” on page 256.

The fill-method approach is designed for simple applications. You declare individual methods by using XML elements named `fill-method`, `sync-method`, and `get-method` in a destination definition in a configuration file. You map these XML elements to methods of any name on your assembler class. For more information, see “[The fill-method approach](#)” on page 264.

Add the methods required for the feature set required for your clients. For example, if your client application is read-only, you only have to provide a get method or a fill method as needed by your clients. A get method is required on an assembler if a client uses the `DataService.getItem()` method. A get method is also required if you set the value of the `cache-items` property to `false` and have lazy associations that point to an item or you are using paging. A fill method is required if a client uses the `DataService.fill()` method.

Later, if you want the assembler to support updates, you add a `sync-method` definition. In this approach, the assembler class does not have to implement any special Flex interfaces.

You can also combine the Assembler interface approach and the fill-method approach in the same assembler class. If you use an XML declaration for a method, the Data Management Service calls that method instead of the equivalent implementation of an Assembler interface method. Combining the Assembler interface approach with fill-method and count-method XML declarations in a destination definition can be a useful way to expose queries to your clients; it prevents the assembler's actual `fill()` method from acting on different types of fill requests. If a matching `fill-method` element is found in the destination, the Data Management Service calls that method. Otherwise, the Data Management Service calls the assembler's implementation of the `fill()` method defined in the Assembler interface.

Note: When creating assemblers for use in LiveCycle Foundation rather than a stand-alone LiveCycle Data Services web application, you must use the Assembler interface approach exclusively.

Configuring a destination that uses a custom assembler

The following table describes the destination properties for using the Java Adapter with a custom assembler. Unless otherwise indicated, these elements are subelements of the `properties` element.

Element		Description
source		Required. Assembler class name.
scope		Optional. Scope of the assembler. Valid values are application, session, and request.
item-class		Optional. For more information, see “ Using strong and anonymous types ” on page 251.
cache-items		Optional. Default value is true. Determines whether client-side DataService caches the last version of each item.
metadata		
	identity	Required. Property to guarantee unique identity among items in a collection of objects. For more information, see “ Uniquely identifying data items ” on page 249.
network		
	paging	Optional. Contains attributes for configuring data paging. For more information, see “ Data paging ” on page 303.
fill-method		(fill-method approach only) Method to invoke when a fill request is made. The required <code>name</code> child element designates the name of the method to call. The optional <code>params</code> element designates input parameter types for the fill operation, and accommodates method overloading. The optional <code>fill-contains-method</code> element points to a method in the Assembler class that takes a changed or created item and the List of fill parameters and returns true or false, depending on whether the item is in that fill. If the fill is not ordered and the method returns <code>true</code> , the item is added to the end of the list of managed items for that fill. If the fill is ordered and you return <code>true</code> , the fill method is re-executed to get the proper ordering of the list. If the method returns <code>false</code> , the fill-method is left as is for that update. The optional <code>ordered</code> element determines whether order is important for this filled collection. Default value is true. Allows for performance optimization when order is not important. The optional <code>security-constraint</code> and <code>security-run-as</code> elements reference a security setting in the <code>services-config.xml</code> file.

Element	Description
sync-method	(fill-method approach only) Method to invoke for update, delete, and insert operations. The required name child element specifies the name of the method to invoke. There is no params element because the parameter is predefined as a List of ChangeObjects. The optional security-constraint and security-run-as child elements reference a security setting in the services-config.xml file.
get-method	(fill-method approach only) Method to retrieve a single item instance instead of a List of item instances. If present, this element always takes a single parameter, which is the Map of values used to denote object identity. The required name child element specifies the name of the method to invoke.
count-method	(fill-method approach only) Method to retrieve an integer from the assembler that you can use to return a count of items without first loading all of a data set into memory. Like the fill-method element, the count-method element can accept parameters. This method does not retrieve the size of a filled sequence, in that any given sequence can represent a subset of data, and sequences are always fully loaded into the server. Count method implementations can execute a COUNT statement against a database. The required name child element specifies the name of the method to invoke.

The following example shows a simple destination definition; it uses an assembler that implements the Assembler interface:

```
<destination id="inventory">
  <properties>
    <source>flex.samples.product.ProductAssembler</source>
    <scope>application</scope>
    <metadata>
      <identity property="productId" />
    </metadata>
    <network>
      <paging enabled="false" pageSize="10" />
    </network>
  </properties>
</destination>
```

The Assembler interface approach

To use the Assembler interface approach in an assembler, you must implement the flex.data.assemblersAssembler interface. The flex.data.assemblers.AbstractAssembler class extends this interface, so extending this class is the easiest way to write your own assembler.

Note: When you create assemblers for use in LiveCycle Foundation rather than a stand-alone LiveCycle Data Services web application, you use the Assembler interface approach exclusively.

Depending on the fill request that a client sends, the `fill()` method performs a specific query to create the Collection object that is returned to the client side. An assembler that implements the Assembler interface has an `autoRefreshFill()` method, which returns a Boolean value that determines whether to refresh fills when data items change. The `autoRefreshFill()` method of the `AbstractAssembler` class returns `true`.

Whenever an item is created or updated, the `autoRefreshFill()` method is called for every fill managed by the server at the time of the create or update. Fills are identified by a set of fill parameters, so the `autoRefreshFill()` method is called once with every set of outstanding fill parameters. If the `autoRefreshFill()` method returns `true` for a given set of fill parameters, the `refreshFill()` method is invoked with those fill parameters, the created or updated item, and a flag denoting whether the change is an update or a create. The purpose of the `refreshFill()` method is to determine if and how the modification of the item affects its membership in that fill.

The `refreshFill()` method can return the following values:

Value	Usage
EXECUTE_FILL	Re-executes a fill for this newly updated or created item. Once the fill is re-executed, the new result is used to determine whether the create or update has caused the membership and position of the modified item to change for the particular fill. This is the default behavior. If you do not override the <code>refreshFill()</code> method in your custom assembler code, this is the behavior it uses.
DO_NOT_EXECUTE_FILL	Do not execute a fill for this newly updated or created item.
APPEND_TO_FILL	Adds the changed item to the end of the list returned by the last fill invocation.
REMOVE_FROM_FILL	Removes this item from the set of items in this filled collection if it is in the collection.

The default behavior of the `refreshFill()` method is to always return `EXECUTE_FILL`. To improve performance, you can override the `refreshFill()` method to base its behavior on the value of the `isCreate` flag. On a create (the `isCreate` flag is `true`), you could return `APPEND_TO_FILL` or `DO_NOT_EXECUTE_FILL` for each set of fill parameters, depending on whether the new item belongs to that fill or not. On an update (the `isCreate` flag is `false`), you could return `APPEND_TO_FILL` if the item was not part of the fill but after the change is part of the fill, `REMOVE_FROM_FILL` if the item was part of the fill but after the change is no longer part of the fill, or `DO_NOT_EXECUTE_FILL` if the update has no impact on fill membership.

In addition to `fill()` methods, the Assembler interface provides `getItem()` and `getItems()` methods for getting specific data items. It also provides the `createItem()` method for creating new items, the `updateItem()` method for updating an existing item, and the `deleteItem()` method for deleting items. You can override the `createItem()`, `updateItem()`, and `deleteItem()` methods to add your own synchronization logic.

The following example shows the source code for the `ProductAssembler` class that is part of the Data Management Service application in the LiveCycle Data Services Test Drive:

```
package flex.samples.product;

import java.util.List;
import java.util.Collection;
import java.util.Map;

import flex.data.DataSyncException;
import flex.data.assemblers.AbstractAssembler;

public class ProductAssembler extends AbstractAssembler {

    public Collection fill(List fillParameters, PropertySpecifier ps) {
        ProductService service = new ProductService();
        return service.getProducts();
    }

    public Object getItem(Map identity) {
        ProductService service = new ProductService();
        return service.getProduct(((Integer) identity.get("productId")).intValue());
    }

    public void createItem(Object item) {
        ProductService service = new ProductService();
        service.create((Product) item);
    }

    public void updateItem(Object newVersion, Object prevVersion, List changes) {
        ProductService service = new ProductService();
        boolean success = service.update((Product) newVersion);
        if (!success) {
            int productId = ((Product) newVersion).getProductId();
            throw new DataSyncException(service.getProduct(productId), changes);
        }
    }

    public void deleteItem(Object item) {
        ProductService service = new ProductService();
        boolean success = service.delete((Product) item);
        if (!success) {
            int productId = ((Product) item).getProductId();
            throw new DataSyncException(service.getProduct(productId), null);
        }
    }
}
```

The Java source code for the ProductAssembler class and other classes it uses are in the WEB_INF/src/flex/samples/product directory of the lcds-samples web application. The ProductAssembler class uses the Assembler interface approach. It extends the flex.data.assemblers.AbstractAssembler class and does not override the `autoRefreshFill()` method, which returns `true`.

ProductAssembler delegates SQL database interaction to a data access object (DAO) called `ProductService`. The following example shows the source code for the `ProductService` class:

```
package flex.samples.product;

import java.util.ArrayList;
import java.util.List;
import java.sql.*;

import flex.samples.ConnectionHelper;
import flex.samples.DAOException;

public class ProductService {

    public List getProducts() throws DAOException {

        List list = new ArrayList();
        Connection c = null;

        try {
            c = ConnectionHelper.getConnection();
            Statement s = c.createStatement();
            ResultSet rs = s.executeQuery("SELECT * FROM product ORDER BY name");
            while (rs.next()) {
                list.add(new Product(rs.getInt("product_id"),
                    rs.getString("name"),
                    rs.getString("description"),
                    rs.getString("image"),
                    rs.getString("category"),
                    rs.getDouble("price"),
                    rs.getInt("qty_in_stock")));
            }
        } catch (SQLException e) {
            e.printStackTrace();
            throw new DAOException(e);
        } finally {
            ConnectionHelper.close(c);
        }
        return list;
    }

    public List getProductsByName(String name) throws DAOException {

        List list = new ArrayList();
        Connection c = null;

        try {
            c = ConnectionHelper.getConnection();
            PreparedStatement ps = c.prepareStatement("SELECT * FROM product WHERE UPPER(name) LIKE ? ORDER BY name");
            ps.setString(1, "%" + name.toUpperCase() + "%");
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                list.add(new Product(rs.getInt("product_id"),
                    rs.getString("name"),
                    rs.getString("description"),
                    rs.getString("image"),
                    rs.getString("category"),
                    rs.getDouble("price"),
                    rs.getInt("qty_in_stock")));
            }
        } catch (SQLException e) {
            e.printStackTrace();
            throw new DAOException(e);
        } finally {
            ConnectionHelper.close(c);
        }
        return list;
    }
}
```

```
        rs.getInt("qty_in_stock"));
    }
} catch (SQLException e) {
    e.printStackTrace();
    throw new DAOException(e);
} finally {
    ConnectionHelper.close(c);
}
return list;
}

public Product getProduct(int productId) throws DAOException {

    Product product = new Product();
    Connection c = null;

    try {
        c = ConnectionHelper.getConnection();
        PreparedStatement ps = c.prepareStatement("SELECT * FROM product WHERE
product_id=?");
        ps.setInt(1, productId);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            product = new Product();
            product.setProductId(rs.getInt("product_id"));
            product.setName(rs.getString("name"));
            product.setDescription(rs.getString("description"));
            product.setImage(rs.getString("image"));
            product.setCategory(rs.getString("category"));
            product.setPrice(rs.getDouble("price"));
            product.setQtyInStock(rs.getInt("qty_in_stock"));
        }
    } catch (Exception e) {
        e.printStackTrace();
        throw new DAOException(e);
    } finally {
        ConnectionHelper.close(c);
    }
    return product;
}

public Product create(Product product) throws DAOException {

    Connection c = null;
    PreparedStatement ps = null;
    try {
        c = ConnectionHelper.getConnection();
        ps = c.prepareStatement("INSERT INTO product (name, description, image, category,
price, qty_in_stock) VALUES (?, ?, ?, ?, ?, ?)");
        ps.setString(1, product.getName());
        ps.setString(2, product.getDescription());
        ps.setString(3, product.getImage());
        ps.setString(4, product.getCategory());
        ps.setDouble(5, product.getPrice());
        ps.setInt(6, product.getQtyInStock());
        ps.executeUpdate();
    }
}
```

```
Statement s = c.createStatement();
// HSQLDB Syntax to get the identity (company_id) of inserted row
ResultSet rs = s.executeQuery("CALL IDENTITY()");
// MySQL Syntax to get the identity (product_id) of inserted row
// ResultSet rs = s.executeQuery("SELECT LAST_INSERT_ID()");
rs.next();

// Update the id in the returned object. This is important as this value must get returned to
the client.
    product.setProductId(rs.getInt(1));
} catch (Exception e) {
    e.printStackTrace();
    throw new DAOException(e);
} finally {
    ConnectionHelper.close(c);
}
return product;
}

public boolean update(Product product) throws DAOException {

    Connection c = null;

    try {
        c = ConnectionHelper.getConnection();
        PreparedStatement ps = c.prepareStatement("UPDATE product SET name=?,
description=?, image=?, category=?, price=?, qty_in_stock=? WHERE
product_id=?");
        ps.setString(1, product.getName());
        ps.setString(2, product.getDescription());
        ps.setString(3, product.getImage());
        ps.setString(4, product.getCategory());
        ps.setDouble(5, product.getPrice());
        ps.setInt(6, product.getQtyInStock());
        ps.setInt(7, product.getProductId());
        return (ps.executeUpdate() == 1);
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DAOException(e);
    } finally {
        ConnectionHelper.close(c);
    }
}

public boolean remove(Product product) throws DAOException {

    Connection c = null;
```

```
try {
    c = ConnectionHelper.getConnection();
    PreparedStatement ps = c.prepareStatement("DELETE FROM product WHERE product_id=?");
    ps.setInt(1, product.getProductId());
    int count = ps.executeUpdate();
    return (count == 1);
} catch (Exception e) {
    e.printStackTrace();
    throw new DAOException(e);
} finally {
    ConnectionHelper.close(c);
}
}

public boolean delete(Product product) throws DAOException {
    return remove(product);
}
}
```

ProductAssembler stores the state of individual products in a JavaBean named Product. The following example shows the source code for the Product class:

```
package flex.samples.product;
import java.io.Serializable;

public class Product implements Serializable {

    static final long serialVersionUID = 103844514947365244L;

    private int productId;
    private String name;
    private String description;
    private String image;
    private String category;
    private double price;
    private int qtyInStock;

    public Product() {

    }

    public Product(int productId, String name, String description, String image, String
category, double price, int qtyInStock) {
        this.productId = productId;
        this.name = name;
        this.description = description;
        this.image = image;
        this.category = category;
        this.price = price;
        this.qtyInStock = qtyInStock;
    }

    public String getCategory() {
        return category;
    }
}
```

```
public void setCategory(String category) {
    this.category = category;
}
public String getDescription() {
    return description;
}
public void setDescription(String description) {
    this.description = description;
}
public String getImage() {
    return image;
}
public void setImage(String image) {
    this.image = image;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public double getPrice() {
    return price;
}
public void setPrice(double price) {
    this.price = price;
}
public int getProductId() {
    return productId;
}
public void setProductId(int productId) {
    this.productId = productId;
}
public int getQtyInStock() {
    return qtyInStock;
}
public void setQtyInStock(int qtyInStock) {
    this.qtyInStock = qtyInStock;
}
}
```

Note: Include the `messaging.jar` and `flex-messaging-common.jar` files in your classpath when you compile code that uses LiveCycle Data Services Java APIs.

The following example shows a destination that uses ProductAssembler:

```
<destination id="inventory">
  <properties>
    <source>flex.samples.product.ProductAssembler</source>
    <scope>application</scope>
    <metadata>
      <identity property="productId"/>
    </metadata>
    <network>
      <paging enabled="false" pageSize="10" />
    </network>
  </properties>
</destination>
```

The fill-method approach

The fill-method approach is designed for simple applications. You declare individual methods by using XML elements named `fill-method`, `sync-method`, and `get-method` in a destination definition in a configuration file. You use these elements to map arbitrary methods of an assembler as fill and sync methods. The method names that you specify in these elements correspond to methods in the assembler class. These methods are called when data operations occur on a Flex client application. Never declare a `sync-method` element if the items are read-only.

If the signature of the fill request sent for a client does not match a `fill-method` element, the Data Management Service calls the `fill()` method defined in the assembler if it implements the `flex.data.assemblersAssembler` interface.

When you use `fill-method` elements, each fill method is identified by the types of parameters declared in a corresponding `fill-method` element. Do not declare two fill methods that take the same types of arguments or an error occurs when you try to invoke the fill method. Another important consideration is that a null value passed to a fill method acts like a wildcard that matches any data type. If you expect to use many fill methods, assign them a name or identity as the first parameter to uniquely identify the query you want to perform. You can then initiate queries with different names within your fill method.

You can implement any number of fill methods to fill a client-side `ArrayCollection` object with data items. You specify fill methods in the `fill-method` section of the destination; each of these methods is assigned as a fill method that can accept an arbitrary number of parameters of varying types and returns a `List` object. The data types must be available at run time. Any call to a client-side `DataService` component's `fill()` method results in a server-side call to the appropriate Java fill method with the parameters that the client provides.

In a `sync-method` element, you specify a method that accepts a change list as its only input parameter. You can choose to implement the `sync` method as a single method that takes a list of `ChangeObject` objects, or you can extend the `AbstractAssembler` class and override the `updateItem()`, `createItem()`, and `deleteItem()` methods.

The change list is a `java.util.List` implementation that contains objects of type `flex.data.ChangeObject`. Each `ChangeObject` in the list contains methods to access an application-specific changed `Object` instance. Each `ChangeObject` also contains convenience methods for getting more information about the type of change that occurred and for accessing the changed data members. Only use the iterator in this list to iterate through the `ChangeObjects`. Use of the `get` method in the list is not allowed. The `flex.data.ChangeObject` class is included in the LiveCycle Data Services Javadoc API documentation.

Note: The compiled assembler class and any other required classes, such as a DAO class, must be available in the web application classpath. When you compile code that uses LiveCycle Data Services Java APIs, you include the `messaging.jar` and `flex-messaging-common.jar` files in your classpath.

Detecting changes to fill results

You use fill methods to implement queries that return a collection of items for a particular destination. When a client executes a fill method with its `DataService.autoSyncEnabled` property set to `true`, it listens for changes to the results of this query so that it can update the list displayed on the client. By default, the Data Management Service implements this functionality by using a technique called *auto-refresh*, which usually ensures that the client results are in sync with the database. With auto-refresh, whenever either the client or the server push API creates or updates an item, the Data Management Service re-executes any `fill()` methods currently managed on active clients. The Data Management Service compares the identity properties of the returned items to its cached results. If any have been added or removed, the Data Management Service builds an update collection message that it sends to the clients that have that fill method cached. For simple applications, auto-refresh could be all you require; however, when the number of fill methods with unique parameters managed by clients grows large, the auto-refresh technique can result in the execution of many unnecessary queries.

You can keep fill methods from being executed unnecessarily in several ways. You can disable auto-refresh, and use the `refreshFill()` method of the `flex.data.DataServiceTransaction` class (the server push API) to explicitly invalidate a fill or set of fill methods. You explicitly invalidate a fill or set of fill methods by specifying a pattern that matches the parameters of the fill methods you want to refresh. Alternatively, you can leave auto-refresh enabled, and set it so that it only re-executes a fill method when a specified item changes the fill results. Another option is to disable auto-refresh and call the `addItemToFill()` or `removeItemFromFill()` methods to make explicit changes to an individual filled collection.

When you leave auto-refresh enabled, you supply a method that is called for each item that is created or updated for each unique set of fill parameters currently managed on active clients. This method returns a value that indicates how the fill should be treated for that change. How this method works depends on whether your assembler implements the Assembler interface or uses XML-based `fill-method` and `sync-method` elements in a destination.

Detecting changes with the Assembler interface approach

When you implement the Assembler interface, a `refreshFill()` method is called for each active fill managed on the client. The `refreshFill()` method is given the list of fill parameters and the changed items and returns a code that indicate whether the fill should be re-executed, the item should be appended to the fill, the fill should be left unchanged for this operation, or the item should be removed from the fill; for more information, see “[The Assembler interface approach](#)” on page 256.

Detecting changes with the fill-method approach

When you use the fill-method approach, you define a `fill-contains-method` element that points to a method in your Assembler class. That method takes the changed or created item and the List of fill parameters and returns `true` or `false` depending on whether the item is in that fill result. If the fill is not ordered and the method returns `true`, the item is added to the end of the list of managed items for that fill. If the fill is ordered and you return `true`, the fill method is re-executed to get the proper ordering of the list. If the method returns `false`, the fill method is left as is for that update.

The following example shows `fill-method` elements with `fill-contains-method` elements for unordered and ordered fills:

```
<fill-method>
    <name>loadUnorderedPatternGroups</name>
    <params>java.lang.String, java.lang.Boolean</params>
    <ordered>false</ordered>
    <fill-contains-method>
        <name>unorderedPatternContains</name>
    </fill-contains-method>
</fill-method>
<fill-method>
    <name>loadOrderedPatternGroups</name>
    <params>java.lang.Boolean, java.lang.String</params>
    <ordered>true</ordered>
    <fill-contains-method>
        <name>orderedPatternContains</name>
    </fill-contains-method>
</fill-method>
```

The following example shows the `unorderedPatternContains()` and `orderedPatternContains()` methods in the corresponding assembler:

```
...
    public boolean unorderedPatternContains(Object[] params, Object group) {
        AssocGroup g = (AssocGroup)group;
        if (g.getName().indexOf((String)params[0]) != -1)
            return true;
        return false;
    }

    public boolean orderedPatternContains(Object[] params, Object group) {
        AssocGroup g = (AssocGroup)group;
        if (g.getName().indexOf((String)params[1]) != -1)
            return true;
        return false;
    }
```

Using property specifiers

There are two basic use cases for the `PropertySpecifier` class. In one case, you receive a `PropertySpecifier` in assembler methods such as `getItem()`. In this case, the `PropertySpecifier` provides a hint so that your code can optimize the fetching of the properties. For the `Assembler.getItem()` method you only need to return the properties specified in the `PropertySpecifier`.

There are a couple of ways to determine which properties are specified in the `PropertySpecifier`. You can use the `PropertySpecifier.includeProperty(String name)` method to test if a specific property is included by a given `PropertySpecifier`. You also can look at how the `PropertySpecifier` is defined.

A `PropertySpecifier` can have a default mode, which means that all properties should be returned unless they are listed as exceptions in the configuration. For example, if you define an association with `load-on-demand` or `page-size`, it is not included in the default `PropertySpecifier`. You can also define a `PropertySpecifier` for the `all` mode, in which case, it returns all properties. `PropertySpecifiers` can also be defined to contain an explicit list of property names, or you can define one that includes the default properties plus an explicit list of property names.

default All properties other than load on demand or paged association properties.

all All properties.

default plus list Default properties plus an extra set of properties retrievable by the `getExtraProperties()` method.

just list A set of properties retrievable by the `getExtraProperties()` method.

A typical reason you might use the PropertySpecifier is if you define master and detail views of the properties in your value objects. All properties in the detail view would be marked as load-on-demand or with page-size so that they are not put into the default set of properties. The only properties you can mark load-on-demand are associations. When the initial `fill()` or `getItem()` call is made for that item, it does not need to return any of these load-on-demand or paged properties. The default `fill()` method must only return properties defined in as default properties for that item. When the client tries to access these properties, a separate request is made to the server to fetch them. In that case, if you set load-on-demand to `true`, your `getItem()` method is called with a PropertySpecifier that only includes the property accessed on the client. If you set the page-size attribute on your association property, the `getPagedCollectionProperty()` method is used instead to fetch a range of items from that association property.

The second use case for a PropertySpecifier is when you must specify one for a Data Management Service API such as the `DataServiceTransaction` class. The PropertySpecifier provides a way to specify a list of properties for a given operation. For example, the `refreshFill()` method takes a PropertySpecifier that specifies which properties from the adapter layer should be refreshed. The `DataServiceTransaction.getItem()` method can also take a PropertySpecifier to indicate which properties should be fetched from the assembler. Whenever you call a Data Management Service API function that takes a PropertySpecifier argument, null can be used instead of providing a PropertySpecifier instance. For the `refreshFill()` method, supplying null means to not refresh any properties of items. In that case, items are only added to or removed from the collection as necessary; no properties of updated items are refreshed. For the `getItem()` method, passing null means to use the default property descriptor.

To create a new PropertySpecifier on the server, you first need to get the `DataDestination` for the type of object you are dealing with. You do this with the static method `DataDestination.getDataDestination(String destName)`. Once you have the `DataDestination`, you can use the `getDefaultPropertySpecifier()` and `getAllPropertySpecifier()` methods to retrieve predefined PropertySpecifiers that refer to the default set or all properties, respectively.

You also can use the `PropertySpecifier.getPropertySpecifier(DataDestination dest, List props, boolean includeDefault)` to create a PropertySpecifier that refers to a specific set of properties you specify in the `props` parameter. You use this method in general when you want to create a PropertySpecifier from an explicit list of properties. In that case, you have two choices: you can include just the list of properties you specify, or you can include use the `includeDefault` parameter to include all default properties in addition to the list of properties specified in the `props` parameter.

Suppose you have a `Person` type with a load-on-demand `addresses` property and you want to call to the `getItem()` method to retrieve a given user's `addresses` property. You could write the following code to get a PropertySpecifier that retrieves just the `addresses` property:

```
ArrayList props = new ArrayList();
props.add("addresses");
PropertySpecifier.getPropertySpecifier(DataDestination.getDataDestination
    ("Person"), props, false);
```

You could write the following code to get a PropertySpecifier that includes all of the non-load-on-demand properties (`firstName`, `lastName`, and so forth) in addition to the `addresses` property:

```
ArrayList props = new ArrayList();
props.add("addresses");
PropertySpecifier.getPropertySpecifier(DataDestination.getDataDestination
    ("Person"), props, true);
```

Using server-side logging with custom assemblers

Custom assemblers log messages through the server-side logging system that is configured in the `services-config.xml` file. To log messages, use the `Service.Data.*` filter pattern or another wildcard pattern. For information about server-side logging, see “[Logging](#)” on page 420.

Standard assemblers

LiveCycle Data Services includes standard assemblers that use the Java adapter.

The SQL Assembler is a specialized assembler that provides a bridge from the Data Management Service to a SQL database management system. Use this type of assembler if your database data model is not complex and you want to expose that data model to MXML without writing Java server-side code.

The two Hibernate assemblers are specialized assemblers that provide a bridge from the Data Management Service to the Hibernate object/relational persistence and query service. Use the Hibernate Assembler or the Hibernate Annotations Assembler if you do not want to write a custom assembler, you require access to database objects, you are using or have used the Hibernate object-relational mapping system, and you are comfortable with writing Java code. You configure the Hibernate Assembler in configuration files; you configure the Hibernate Annotations Assembler in Hibernate annotations in Java classes.

The Model Assembler extends the Hibernate Assembler and is used with Adobe application modeling technology; Model Assembler instances are dynamically generated based on a model.

The SQL Assembler

The SQL Assembler is a specialized Java assembler class that you use with the Java adapter to provide a bridge to a SQL database management system. Using the SQL Assembler, you can build simple create, read, update, and delete (CRUD) applications based on the Data Management Service without writing server-side code. You can directly specify the SQL statements that you want to execute when the user creates, updates, or deletes items. You can also specify a number of SELECT statements to retrieve data in different ways. This assembler is useful when your database data model is not very complex and you want to expose it to MXML without writing Java code. You configure the connection to the database and write SQL code in a Data Management Service destination definition. You are only required to include SQL code for operations that a Flex application calls. The SQL Assembler does not support hierarchical (nested) destinations.

The SQL Assembler class, `flex.data.assemblers.SQLAssembler`, extends the `flex.data.assemblers.AbstractAssembler` class and is included in the public LiveCycle Data Services Javadoc API documentation. When necessary, you can extend `SQLAssembler` to override methods. The resources directory the LiveCycle Data Services installation includes the source code for the `SQLAssembler` and `HibernateAssembler` classes.

Configuring a destination that uses the SQL Assembler

Before using the SQL Assembler from a Flex client, you configure a destination that specifies database connection information and SQL statements. The following example shows a complete destination definition that uses the SQL Assembler; the `lcds-samples` web application included with LiveCycle Data Services uses this destination:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>
    <properties>
        <use-transactions>true</use-transactions>
        <source>flex.data.assemblers.SQLAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="PRODUCT_ID"/>
        </metadata>
        <server>
            <database>
                <driver-class>org.hsqldb.jdbcDriver</driver-class>
                <!-- Modify the URL below with the actual location of the
                    flexdemodb database on your system -->
                <url>jdbc:hsqldb:hsq://myserver:9002/flexdemodb</url>
                <username>sa</username>
                <password></password>
                <login-timeout>15</login-timeout>
            </database>

            <!-- Use this syntax when using a JNDI data source-->
            <!--
            <database>
                <datasource>java:comp/env/jdbc/flexdemodb</datasource>
            </database>
            -->

            <actionscript-class>Product</actionscript-class>
            <fill>
                <name>all</name>
                <sql>SELECT * FROM PRODUCT ORDER BY NAME</sql>
            </fill>

            <fill>
                <name>by-name</name>
                <sql>SELECT * FROM PRODUCT WHERE NAME LIKE CONCAT('%',
                    CONCAT(#searchStr#, '%'))</sql>
            </fill>

            <fill>
                <name>by-category</name>
                <sql>SELECT * FROM PRODUCT WHERE CATEGORY LIKE CONCAT('%',
                    CONCAT(#searchStr#, '%'))</sql>
            </fill>

            <get-item>
                <sql>SELECT * FROM PRODUCT WHERE PRODUCT_ID = #PRODUCT_ID#</sql>
            </get-item>

            <create-item>
                <sql>INSERT INTO PRODUCT
                    (NAME, CATEGORY, IMAGE, PRICE, DESCRIPTION, QTY_IN_STOCK)
                    VALUES (#NAME#, #CATEGORY#, #IMAGE#, #PRICE#, #DESCRIPTION#,
                        #QTY_IN_STOCK#)</sql>
                <id-query>CALL IDENTITY()</id-query> <!-- HSQLDB syntax to
                    retrieve value of autoincremented column -->
            </create-item>
        </server>
    </properties>
</destination>
```

```
</create-item>

<update-item>
    <sql>UPDATE PRODUCT SET NAME=#NAME#, CATEGORY=#CATEGORY#,
        IMAGE=#IMAGE#, PRICE=#PRICE#, DESCRIPTION=#DESCRIPTION#,
        QTY_IN_STOCK=#QTY_IN_STOCK#
        WHERE PRODUCT_ID=#_PREV.PRODUCT_ID#</sql>
</update-item>

<delete-item>
    <sql>DELETE FROM PRODUCT WHERE PRODUCT_ID=#PRODUCT_ID#</sql>
</delete-item>

<count>
    <name>all</name>
    <sql>SELECT count(*) FROM PRODUCT</sql>
</count>
</server>
</properties>
</destination>
```

For examples of MXML applications that perform CRUD and query operations on this destination, see the samples web application included with LiveCycle Data Services.

Specifying the Java adapter

Specify an instance of the Java adapter if it is not already set as the default adapter. In the following example, `java-dao` is the `id` value of a Java adapter instance defined in the configuration file:

```
<destination id="SqlPerson">
    <adapter ref="java-dao"/>
    ...
</destination>
```

Setting the destination source

Specify the `flex.data.assemblers.SQLAssembler` class name in the `source` element of the destination definition, as the following examples shows. Generally, you also set the `scope` value to `application`.

```
<destination id="SqlPerson">
    <adapter ref="java-dao"/>
    <properties>
        <use-transactions>true</use-transactions>
        <source>flex.data.assemblers.SQLAssembler</source>
        <scope>application</scope>
    ...
    </properties>
</destination>
```

Setting the identity property

Specify the `identity` property, which maps to the database field. Additionally, this name corresponds to the name of a property of an ActionScript object in the Flex client application. The ActionScript object can be anonymous or strongly typed. The following example shows the configuration for an `identity` element:

```
<destination id="SqlPerson">
    <adapter ref="java-dao"/>
    <properties>
    ...
        <metadata>
            <identity property="PRODUCT_ID"/>
        </metadata>
    ...
    </properties>
</destination>
```

Configuring the database connection

You can connect to a SQL database by specifying a data source or a driver in the destination definition. Using a data source is recommended because it provides application server enhancements, such as connection pooling.

The following example shows a datasource element in a destination definition:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
            <database>
                <driver-class>org.hsqldb.jdbcDriver</driver-class>
                <!-- Modify the URL below with the actual location of the
                    flexdemodb database on your system -->
                <url>jdbc:hsqldb:hsq://yourserver:9002/flexdemodb</url>
                <username>sa</username>
                <password></password>
                <login-timeout>15</login-timeout>
            </database>
        ...
    </properties>
</destination>
```

You use the optional `login-timeout` property for a data source or a driver. You specify this value in seconds. When you use a data source, this value is used as a parameter to the `DataSource.setLoginTimeout(seconds)` method. If no timeout is specified, a timeout is not set and you rely on the data source/application server settings. The timeout settings are almost always included with the data source definition. When you use a driver, set this value as a parameter to the `DriverManager.setLoginTimeout(seconds)` method. If no timeout is specified, the default value is 20 seconds.

The following example shows a database element that provides database driver settings in a destination definition.

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
            <database>
                <datasource>java:comp/env/jdbc/flexdemodb</datasource>
            </database>
        ...
    </properties>
</destination>
```

Configuring return types

For operations that return objects to the Flex client, you can specify the return type in `javaclass-` or `actionscript-class` elements, as the following example shows:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
        ...
            <actionscript-class>Product</actionscript-class>
            <!--<java-class>samples.Product</java-class> -->
        ...
    </server>
    ...
</properties>
</destination>
```

The `actionscript-class` element specifies the ActionScript type that you want returned from the server when no alias mapping to a specific Java class exists. The ActionScript class that you specify must have the `RemoteClass` metadata syntax, but without an alias that specifies a Java class.

The `java-class` element specifies a server-side Java class that maps to an ActionScript class that uses the `RemoteClass(alias=classname)` metadata syntax. You use this element when you map a specific ActionScript type to a specific Java type that uses this syntax. The Java and ActionScript property names match the database field names. The ActionScript class designating the Java class specified in the `java-class` element must be available on the client.

Configuring fill operations

You configure fill operations in `fill` elements in the `server` section of a destination definition. The `fill` elements contain `name` and `sql` or `statement` elements, depending on whether you are using a single SQL statement or a stored procedure. The SQL code indicates properties from the object to use with `#propName#`. The following example shows several `fill` elements:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
        ...
            <fill>          <name>all</name>
                <sql>SELECT * FROM PRODUCT ORDER BY NAME</sql>
            </fill>

            <fill>
                <name>by-name</name>
                <sql>SELECT * FROM PRODUCT WHERE NAME LIKE CONCAT('%',
                    CONCAT(#searchStr#, '%'))</sql>
            </fill>

            <fill>
                <name>by-category</name>
                <sql>SELECT * FROM PRODUCT WHERE CATEGORY LIKE CONCAT('%',
                    CONCAT(#searchStr#, '%'))</sql>
            </fill>
        ...
    </server>
    ...
</properties>
</destination>
```

In a Flex client application, the `DataService.fill()` method uses the named fill query that is defined in the destination. The arguments can be an anonymous ActionScript object with named parameters or a strongly typed ActionScript object. These arguments are used to define the SQL parameter values. A list is returned. The return type can be anonymous, based on a `javaclass-` value, or based on an `actionscript-class` value.

The following example is a simple client-side `DataService.fill()` method without named parameters:

```
ds.fill(dataCollection, named-sql-query, parameters);
```

The following example is a client-side `DataService.fill()` method with named parameters:

```
<mx:Button label="Search" click="ds.fill(products, combo.selectedIndex == 0 ? 'by-name' : 'by-
category', {searchStr: searchStr.text})"/>
```

Configuring a getItem operation

You configure a `getItem` operation in a `get-item` element in the `server` section of a destination definition. The `get-item` element contains a `sql` element or a `statement` element, depending on whether you are using a single SQL statement or a stored procedure. The SQL code indicates properties from the object to use with `#propName#` text, where `propName` is the property name. The following example shows a `get-item` element:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>
    <properties>
        ...
        <server>
            ...
            <get-item>
                <sql>SELECT * FROM PRODUCT WHERE PRODUCT_ID = #PRODUCT_ID#</sql>
            </get-item>
        ...
        </server>
    </properties>
</destination>
```

In a Flex client application, the `DataService.getItem()` method takes a single argument that is the identity of the item. The identity maps to the `identity` property specified in the Data Management Service destination. This field also links with the database field and the property name on the ActionScript object. The `PreparedStatement` uses the `id` argument value for its single ordered parameter.

Note: Do not use a SQL variable name in the `get-item` SQL statement that is different from the `identity` property name. If you do that, when a conflict occurs, the `identity` property is used to call `DataService.getItem()`, resulting in an error because of a parameter mismatch between the `identity` and the SQL variable.

The item returned is an anonymous object with property names consistent with database field names, a strongly typed ActionScript object where the type was specified using the `actionscript-class` element in the destination, or a strongly typed ActionScript object where the `RemoteClass(alias="java-class")` is equivalent to the class specified using the `java-class` element in the destination. In all cases, the property names are consistent with the database field names.

The following example shows the code for a `DataService.getItem()` method:

```
ds.getItem({employee_id:cursor.current.employeeid});
```

The property name `employee_id` is the `id` property on the ActionScript object as well as the database field name.

Configuring a `createItem` operation

You configure a `createItem` operation in a `create-item` element in the `server` section of a destination definition. The `create-item` element contains one or more `sql` or `statement` elements, depending on whether you are using a single SQL statement or a stored procedure. The SQL code indicates properties from the object to use with `#propName#`.

The `createItem` operation requires that you define two SQL queries. The `create-item` element creates the object and the `id-query` element obtains the `identity` property of the newly created object from the database. The `idquery`-element can precede the `create-item` SQL statement. The Data Management Service executes the `sql-update` and the `id-query` within the same `DataServiceTransaction` object and uses the same connection to the database. The parameters for the SQL statement are retrieved from the properties of the `createItem` instance. As an alternative, you can set up a procedure call in the database that does the insert and id selection. For the `create-item` element only, you can designate the `procedure-param` element as an OUT parameter, which means that the value is returned to the client in the `property-value` field of the ActionScript instance.

You can use the identity results of the `id-query` in any SQL statement that comes after it. If the `id-query` comes first, it is executed before all of the SQL statements. If the `id-query` comes after any SQL statement, it is always executed after the first SQL statement. If you require additional flexibility in ordering, use run-time configuration of destinations as described in “[Run-time configuration](#)” on page 411.

The following example shows two `create-item` elements. The second element uses `procedure-param` elements.

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
        ...
<create-item>
    <sql>INSERT INTO employee (first_name, last_name, title, email, phone,
        company_id) VALUES (#firstName#, #lastName#, #title#, #email#,
        #phone#, #companyId#)</sql>
    <id-query>SELECT LAST_INSERT_ID()</id-query>
</create-item>

<create-item>
    <procedure name="MY_INSERT_PROC">
        <procedure-param property-value="#id#" type="OUT"/>
        <procedure-param property-value="#firstName#"/>
        <procedure-param property-value="#lastName#"/>
        <procedure-param property-value="#title#"/>
        <procedure-param property-value="#email#"/>
        <procedure-param property-value="#phone#"/>
        <procedure-param property-value="#companyId#"/>
    </procedure>
</create-item>
...
        </server>
    </properties>
</destination>
```

The following example shows Flex client code for an Employee object and a `DataService.create()` method that creates the employee in the database:

```
...
var employee:Employee = new Employee();
employee.first_name = "Joe";
employee.last_name = "Dev";
employee.title = "engineer";
employee.email = "joedev@adobe.com";
employee.phone = "617-229-2065";
employee.company_id = 2;
ds.createItem(employee);
ds.commit();
...
```

Configuring a deleteItem operation

You configure a `deleteItem` operation in a `delete-item` element in the `server` section of a destination definition. The `delete-item` element contains a `sql` element or a `statement` element, depending on whether you are using a single SQL statement or a stored procedure. The SQL code indicates which properties from the object to use with `#propName#`.

The following example shows a `delete-item` element:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
        ...
            <delete-item>
                <sql>DELETE FROM PRODUCT WHERE PRODUCT_ID=#PRODUCT_ID#</sql>
            </delete-item>
        ...
        </server>
    </properties>
</destination>
```

The following example shows Flex client code for deleting an item from the database. The properties for the instance to delete are mapped to the fields in the `delete-item` query in the configuration file. If all properties are not set, the delete operation fails.

```
...
var removedItem:Object = cursor.remove();
ds.commit();
...
```

If the server value is changed, the delete SQL operation fails to execute. In this situation, a `DataSyncException` is thrown. The exception contains the results of the `get-item` query.

Configuring an updateItem operation

You configure an `updateItem` operation in an `update-item` element in the `server` section of a destination definition. The `update-item` element contains a `sql` element or a `statement` element, depending on whether you are using a single SQL statement or a stored procedure. The SQL code indicates properties from the object to use with `#propName#`.

The following example shows an `update-item` element:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>

    <properties>
    ...
        <server>
        ...
            <update-item>
                <sql>UPDATE PRODUCT SET NAME=#NAME#, CATEGORY=#CATEGORY#,
                    IMAGE=#IMAGE#, PRICE=#PRICE#, DESCRIPTION=#DESCRIPTION#,
                    QTY_IN_STOCK=#QTY_IN_STOCK#
                    WHERE PRODUCT_ID=#_PREV.PRODUCT_ID#</sql>
            </update-item>
        ...
        </server>
    </properties>
</destination>
```

The server passes in the current value and the previous value from the client. If these are different, the parameters of the SQL statement are replaced and the update against the database is executed. If these are different, then the server value could have changed. The item is retrieved from the database using the `get-item` query defined in the destination. You can use `_PREV` to reference the previous version of the object.

The following example shows the Flex client-side ActionScript code for an updateItem operation:

```
...
cursor.current.last_name = lastName.text;
cursor.current.first_name = firstName.text;
ds.commit();
...
```

Configuring a count operation

You configure a count operation in a `count` element in the `server` section of a destination definition. The `count` element contains a `name` element and a `sql` element or a `statement` element, depending on whether you are using a single SQL statement or a stored procedure. The SQL code indicates properties from the object to use with `#propName#`.

The following example shows a `count` element:

```
<destination id="sql-product">
    <adapter ref="java-dao"/>
    <properties>
    ...
        <server>
            ...
                <count>
                    <name>all</name>
                    <sql>SELECT count(*) FROM PRODUCT</sql>
                </count>
            </server>
        </properties>
    </destination>
```

In a Flex client application, the `DataService.count()` method execution and definition are similar to those of the `DataService.fill()` method. You can define multiple SQL statements in the definition. The statement that you use, and its parameters, are passed during the `count` method execution.

The following example shows a simple use of the `DataService.count()` method:

```
ds.count(named-sql-query, parameters);
```

The following example shows a more complex use of the `DataService.count()` method:

```
var countToken:AsyncToken = ds.count("firstNamedCount", parameters);
public function firstNamedCountResultHandler(event):void
{
    var count:int = event.result;
}
```

Using server-side logging with the SQL Assembler

The SQL Assembler logs messages through the server-side logging system that is configured in the `services-config.xml` file. To log SQL Assembler messages, use the `Service.Data.SQL` filter pattern or a wildcard pattern. For information about server-side logging, see “[Logging](#)” on page 420.

The Hibernate assemblers

The Hibernate assemblers are Java assembler classes that you use with the Java adapter to provide a bridge to a Hibernate object/relational persistence and query service. Hibernate provides a persistence mechanism to facilitate data access with a server-side database. There are two Hibernate assemblers, the Hibernate Assembler and the Hibernate Annotations Assembler. You configure the Hibernate Assembler in configuration files. You configure the Hibernate Annotations Assembler in Hibernate annotations.

You create Hibernate mappings to describe how the data represented by a Java object maps to the columns of a database table. LiveCycle Data Services lets you define mappings in two ways: by using Hibernate mapping files or by using Hibernate annotations.

A *mapping file* is an XML file that you deploy on your server with a Java class file. *Annotations* are metadata that you insert in a Java source code file so that they are compiled into the Java class file. For either of these mapping types, you use the Hibernate global configuration file to specify connection information to the database associated with the Java object.

After you define your mappings and create the Hibernate configuration file, you create a LiveCycle Data Services destination that references a Hibernate Assembler. The assembler class that you use depends on how you define your mappings. Use the `HibernateAssembler` class with a mapping file and the `HibernateAnnotationsAssembler` class with annotations.

The Data Management Service `count()`, `get()`, `create()`, `update()`, and `delete()` methods correspond to Hibernate operations of similar names and behavior, and they require no configuration other than the configuring the Hibernate mappings. Any named queries you define in your Hibernate configuration file are directly exposed as queries to Flex clients with no additional configuration as long as the user is authorized to execute queries for this destination based on any read-security-constraint you have set.

For more information on Hibernate, see the Hibernate website at <http://www.hibernate.org>.

Configuring your system for Hibernate

By default, a LiveCycle Data Services web application is not configured to support Hibernate. Before you can run an application that uses Hibernate, copy all JAR files from `install_root]/resources/hibernate` to the WEB-INF/lib directory of your web application.

The Hibernate Assembler supports Hibernate 3.0 and later. Using this assembler, you do not need to write Data-Management-specific Java code to integrate with Hibernate.

Hibernate configuration files

Hibernate uses two types of configuration files: the global configuration file and mapping files. The global configuration file, `hibernate.cfg.xml`, contains declarations for database settings and behaviors. The mapping file contains declarations for how Java classes and fields map to database tables and columns.

Note: You do not use mapping files if you use Hibernate annotations.

Hibernate requires that the files be accessible in the web application classpath. Place the `hibernate.cfg.xml` file in the WEB-INF/classes directory and any mapping files in the same directory as the corresponding Java class file.

The following example `hibernate.cfg.xml` file defines the access to a database and references a single annotated Java file named `Contact`:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings. -->
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">
      jdbc:hsqldb:hsqldb://localhost:9002/flexdemodb
    </property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>

    <!-- JDBC connection pool (use the built-in). -->
    <property name="connection.pool_size">1</property>

    <!-- SQL dialect. -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- Enable the Hibernate automatic session context management. -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache. -->
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Echo all executed SQL to stdout. -->
    <property name="show_sql">true</property>

    <!-- Drop and recreate schema -->
    <property name="hbm2ddl.auto">create</property>

    <!-- Load the annotated Java class. -->
    <mapping class="flex.samples.contactannotate.Contact"/>
  </session-factory>
</hibernate-configuration>
```

Supported Hibernate data retrieval methods

The Hibernate assemblers currently support the following ways to retrieve data from Hibernate:

Hibernate feature	Description
load and get	<p>Retrieves a single object by identity.</p> <p>The Data Management Service <code>getItem()</code> functionality corresponds to the Hibernate <code>get</code> functionality. The Hibernate Assembler does not use the <code>load</code> method.</p>
Hibernate Query Language (HQL)	<p>Retrieves a list of items based on an HQL query.</p> <p>You can configure the Data Management Service <code>fill</code> functionality to recognize <code>fill</code> arguments as HQL query strings along with a map of parameters.</p> <p>To use HQL queries from your client code, set <code><allowhqlqueries>true</allowhqlqueries></code>---- in the <code>fill-configuration</code> section.</p> <p>Note: By allowing clients to specify arbitrary HQL queries, you are exposing a potentially dangerous operation to untrusted client code. Clients may be able to retrieve data you did not intend for them to retrieve or execute queries that cause performance problems in your database. Adobe recommends that you use the HQL queries for prototyping, but replace them with named queries before you deploy your application, and use the default setting <code><allowhqlqueries>false</allow-hql-queries></code>-- in your production environment.</p> <p>When <code>allow-hql-queries</code> is set to <code>true</code>, the first parameter to the <code>fill()</code> method is the string token "flex:hql", the second parameter is the HQL string and the third parameter specifies the parameter set for the queries as either an array or a <code>java.util.Map</code>. The client-side <code>DataService.fill()</code> method looks similar to the following example:</p> <pre>myService.fill(myCollection, "flex:hql", "from Person p where p.firstName = :firstName", {"firstName": "Paul"});</pre> <p>The last parameter contains the parameter name and value bindings for the query. The preceding example uses named parameters.</p> <p>Positional parameters are also supported; that is, you could write the HQL string as follows:</p> <pre>myService.fill(myCollection, "flex:hql", "from Person p where p.firstName = ?", ["Paul"]);</pre> <p>In that case, the last parameter is an array of values instead of a map of values. The use of positional parameters instead of named parameters is considered a less reliable approach, but both approaches are supported.</p>
Named Hibernate queries	<p>In Hibernate, you configure a named query by using the Hibernate configuration element named <code>query</code>. You can then call this query from a Flex client by passing the name of the Hibernate query as the second argument to a <code>DataService.fill()</code> method. As with HQL queries, if the Hibernate query is using named parameters, you specify the parameters on the client with an anonymous object that contains those parameters. If your Hibernate query is using positional parameters, you provide an array of parameters. The following example shows a <code>fill()</code> method that uses a named query:</p> <pre>myDS.fill(myCollection, "eg.MyHibernateEntity.myQueryName", myQueryParameterObjectOrArray);</pre>

About the Hibernate assemblers

The Hibernate assemblers are specialized Java assembler classes that you use with the Java adapter to provide a bridge to the Hibernate object/relational persistence and query service. The Hibernate Assemblers use the Hibernate configuration files at run time to help define the data model and use Hibernate APIs to persist data changes into a relational database. Hibernate operations are encapsulated within the assembler.

The Hibernate assemblers uses the Hibernate mappings to expose named queries to client-side code. Your client code can provide parameters to these queries by using fill parameter arguments. You can also optionally allow clients to use HQL queries by setting the `allow-hql-queries-` element to `true` in the destination configuration.

Note: If you use the `allow-hql-queries` option, you expose arbitrary queries that can be executed by untrusted code. Use this option only for prototyping and switch to named queries before deploying your production application.

Flex supports Hibernate `version` or `timestamp` elements for automatic concurrent conflict detection in Hibernate entity mappings. If these settings are used in a Hibernate configuration file, they take precedence over any update/delete conflict modes defined. A warning is logged that the automatic Hibernate conflict detection is used if conflict modes are defined.

The Hibernate assemblers does not support criteria-based queries but you can extend the `flex.data.assemblers.HibernateAssembler` class and expose whatever queries you like by writing some additional Java code. This class is documented in the LiveCycle Data Services Javadoc API documentation and the source code for the `HibernateAssembler` is included with the product.

To expose custom handling of type mappings on query parameter bindings, Flex honors the Hibernate type mappings provided in the Hibernate configuration file for a persistent Java object. The values for the `type` attribute are treated as Hibernate types that map Java types to SQL types for all supported databases.

The basic steps that you use to integrate Hibernate into your application are as follows:

- 1 Create mappings, either using mapping files or annotations. For more information, see “[Using Hibernate mapping files](#)” on page 281 and “[Using Hibernate annotations](#)” on page 284.
- 2 Edit the global Hibernate configuration file, `hibernate.cfg.xml`, to define the database connection and to specify the Java classes that support Hibernate. For more information, see “[Hibernate configuration files](#)” on page 278.
- 3 Define an adapter that uses the `JavaAdapter`. To use the Hibernate Assembler, specify an instance of the Java adapter. For more information, see “[Using Hibernate mapping files](#)” on page 281.
- 4 Define a LiveCycle Data Services destination that references the Hibernate Assembler in the `data-management-config.xml` file. For more information, see “[Configuring destinations to use Hibernate](#)” on page 290.

Using Hibernate mapping files

A Hibernate mapping file contains declarations for how Java classes and fields map to database tables and columns. The Data Management Service must have access to this file at run time. Hibernate requires that the file be accessible in the web application classpath.

Example using Hibernate mapping files

The Test Drive example in the `lcds-samples\hibernate` directory uses a Hibernate mapping file with a Java class. In that example, the Java class is named `Contact.java`, and the associated mapping file is named `Contact.hbm.xml`. When you deploy the compiled `Contact.class` file, you deploy the mapping file to the same directory. In this example, the deployment directory is `lcds-samples\WEB-INF\classes\flex\samples\contact`.

You can view the application by requesting the following URL:

`http://localhost:8400/lcds-samples/hibernate/index.html`

Note: Before you run this application, make sure that you configure your system as described in the section “[Configuring your system for Hibernate](#)” on page 278, and remove the comments from the `hibernate-contact` destination in the `lcds-samples\WEB-INF\flex\data-management-config.xml` file.

The following example shows the mapping file, `Contact.hbm.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="flex.samples.contact.Contact" table="contact">
    <id name="contactId" column="contact_id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name"/>
    <property name="lastName" column="last_name"/>
    <property name="address"/>
    <property name="city"/>
    <property name="state"/>
    <property name="zip"/>
    <property name="country"/>
    <property name="phone"/>
    <property name="email"/>
    <property name="notes"/>
  </class>
  <query name="all">From Contact</query>
</hibernate-mapping>
```

The Contact.hbm.xml file contains the following information:

- The `class` element specifies the name of the associated Java class and database table.
- The `id` element specifies the property that corresponds to the primary key of the database table.
- The `property` elements associate a property of the Java class with a column in the database table. Use the `column` attribute with the `firstName` and `lastName` properties to specify the column in the database table corresponding to those properties. This configuration is necessary because the column name does not match the property name. None of the remaining properties requires the `column` attribute because the column name in the database table matches the property name.
- The `query` element specifies the named query used to populate the Java object from the database.

To connect to the Java class and use the associated Hibernate mappings, edit the `data-management-config.xml` file to define an adapter that uses the `JavaAdapter` and a destination that uses the `HibernateAssembler`. The following example shows an excerpt from the `data-management-config.xml` file:

```
<adapters>
  <adapter-definition id="actionscript"
    class="flex.data.adapters.ASObjectAdapter"
    default="true"/>
  <!-- Define an adapter that uses the JavaAdapter. -->
  <adapter-definition id="java-dao"
    class="flex.data.adapters.JavaAdapter"/>
</adapters>

<destination id="hibernate-contact">
  <!-- Reference the JavaAdapter. -->
  <adapter ref="java-dao"/>
  <properties>
    <use-transactions>true</use-transactions>
    <!-- Specify the HibernateAssembler. -->
    <source>flex.data.assemblers.HibernateAssembler</source>
    <scope>application</scope>
    <network>
      <paging enabled="false" pageSize="10" />
    </network>
    <server>
      <!-- Specify the Java class. If this is not specified,
          it defaults to the destinationId-->
      <item-class>flex.samples.contact.Contact</item-class>
      <fill-configuration>
        <use-query-cache>false</use-query-cache>
        <allow-hql-queries>true</allow-hql-queries>
      </fill-configuration>
    </server>
  </properties>
</destination>
```

Use the `metadata` element to specify managed association elements, such as the `page-size`, `load-on-demand`, `one-to-many`, `many-to-one`, or other options that differ from the default determined from the hibernate configuration. For more information on the `metadata` element, see “[Hierarchical data](#)” on page 293.

The `<identity>` child element of the `metadata` element specifies the primary key of the object. When working with Hibernate Assemblers, the `identity` element is optional. If you omit the `identity` element, the `metadata` section is automatically generated from the Hibernate configuration.

If you include the `<identity>` element, the `property` attribute must specify the name of a property in the Java class. In this case, no default information is generated. Include the `identity` element when you want to take complete control over the metadata section for your data type. You can then handle Hibernate associations without creating associations in Data Management so that they are treated as value objects. This technique is an advanced use and is not typically recommended.

Finally, create an instance of the DataService component in your client-side Flex application that references the Hibernate destination:

```
<mx:DataService id="ds"
  destination="hibernate-contact"
  conflict="conflictHandler(event)"
  autoCommit="false"/>
```

Using Hibernate annotations

Hibernate annotations replace the Hibernate mapping file with metadata that you add to the Java source code file. Annotations simplify your implementation because you do not have to maintain the mapping file separately from the source code file. Instead, you include mapping information and source code in a single Java file.

Add annotations to your Java class

The general steps for using Hibernate annotations in a Java file are as follows:

- 1 Ensure that your system is configured to use Hibernate annotations as described in “[Configuring your system for Hibernate](#)” on page 278.

- 2 Edit your Java class to import javax.persistence.*.

```
package flex.samples.contactannotate;

// Import JPA annotations.
import javax.persistence.*;
...
```

- 3 Add annotations to the Java class.

- 4 Compile the class, making sure to include ejb3-persistence.jar in the classpath. The ejb3-persistence.jar contains the javax.persistence package. You copied ejb3-persistence.jar file to the WEB-INF/lib directory when you copied all JAR files from install_root]/resources/hibernate to the WEB-INF/lib.

- 5 Edit hibernate.cfg.xml to add a mapping to the class file, as the following example shows:

```
<!-- Load the annotated Java class. -->
<mapping class="flex.samples.contactannotate.Contact"/>
```

Notice that you use the `class` attribute of the `mapping` element to specify the annotated class, rather than the `resource` attribute to specify a mapping file.

- 6 Edit the data-management-config.xml file to define a destination that uses the `HibernateAnnotationsAssembler` and specifies the Java class as the value of the `item-class` element.

Note: If you change anything in data-management-config.xml, you typically have to recompile your client-side applications and restart your server-side application for the changes to take effect.

Example using Hibernate annotations

The following example modifies the Test Drive example in the lcds-samples/hibernate directory to replace the Hibernate mapping file with annotations in the Java class. This example also shows how to modify the hibernate.cfg.xml and data-management-config.xml files, and the Contact.as file of the client-side Flex code.

Create Contact.java

- 1 Create a directory for the annotated Java class named lcds-samples\WEB-INF\src\flex\samples\contactannotate.
- 2 Copy Contact.java from lcds-samples\WEB-INF\src\flex\samples\contact to contactannotate.
- 3 Edit contactannotate\Contact.java to perform the following:
 - Rename the package to flex.samples.contactannotate to correspond to the new directory name.
 - Add the `import` statement for the javax.persistence package.
 - Add the `@Entity` and `@Table` annotations to specify the name of the database table.
 - Add the `@NamedQuery` annotation to specify the query mapping for the database.

- Add the @ID and @Column annotations to specify the property that corresponds to the primary key of the database table.
- Add the @Column annotations to the firstName and lastName properties to specify the column in the database table corresponding to those properties. This configuration is necessary because the column name does not match the property name. None of the remaining properties requires the @Column annotation because the column name in the database table matches the property name.

The following code shows the modified Contact.java file:

```
package flex.samples.contactannotate;

// Import JPA annotations.
import javax.persistence.*;

// Map the @Entity class to the contact database table.
@Entity
@Table(name="contact")
// Create a query mapping.
@NamedQuery(name="all",query="From Contact")
public class Contact {

    // Map the identifier property to the contactId column.
    @Id @GeneratedValue
    @Column(name="contact_id")

    private int contactId;
    @Column(name="first_name")

    private String firstName;
    @Column(name="last_name")

    private String lastName;

    private String address;

    private String city;

    private String state;

    private String zip;

    private String country;

    private String email;

    private String phone;

    private String notes;

    public String getAddress() {

        return address;
```

```
}

public void setAddress(String address) {

    this.address = address;
}

public String getCity() {

    return city;
}

public void setCity(String city) {

    this.city = city;
}

public int getContactId() {

    return contactId;
}

public void setContactId(int contactId) {

    this.contactId = contactId;
}

public String getCountry() {

    return country;
}

public void setCountry(String country) {

    this.country = country;
}

public String getEmail() {

    return email;
}
```

```
public void setEmail(String email) {  
  
    this.email = email;  
}  
  
public String getFirstName() {  
  
    return firstName;  
}  
  
public void setFirstName(String firstName) {  
  
    this.firstName = firstName;  
}  
  
public String getLastName() {  
  
    return lastName;  
}  
  
public void setLastName(String lastName) {  
  
    this.lastName = lastName;  
}  
  
public String getNotes() {  
  
    return notes;  
}  
  
public void setNotes(String notes) {  
  
    this.notes = notes;  
}  
  
public String getPhone() {  
  
    return phone;  
}
```

```
public void setPhone(String phone) {  
  
    this.phone = phone;  
}  
  
public String getState() {  
  
    return state;  
}  
  
public void setState(String state) {  
  
    this.state = state;  
}  
  
public String getZip() {  
  
    return zip;  
}  
  
public void setZip(String zip) {  
  
    this.zip = zip;  
}
```

4 Change directory to lcds-samples.

5 Compile the class by using the following javac command:

```
javac -d WEB-INF/classes/ -cp WEB-INF/lib/ejb3-persistence.jar  
WEB-INF/src/flex/samples/contactannotate/Contact.java
```

Edit hibernate.cfg.xml

The global Hibernate configuration file, hibernate.cfg.xml, specifies many Hibernate configuration options, including the `mapping` property. Edit the lcds-samples\WEB-INF\classes\hibernate.cfg.xml file to perform the following:

- Use the `mapping` property and `class` attribute to specify the annotated Java class.
- Comment out the reference to the Contact mapping file.

The following code shows the hibernate.cfg.xml for this example, with the new `mapping` property:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">
            jdbc:hsqldb:hsq1://localhost:9002/flexdemodb
        </property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable the Hibernate automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache -->
        <property name="cache.provider_class">
            org.hibernate.cache.NoCacheProvider
        </property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and recreate schema -->
        <property name="hbm2ddl.auto">create</property>

        <!-- Add these lines to reference the annotated java file. -->
        <!-- Load the annotated Java class. -->
        <mapping class="flex.samples.contactannotate.Contact"/>

        <!-- Add comments around these lines to remove the reference to the mapping file. -->
        <!-- Load the database table mapping file -->
        <!-- <mapping resource="flex/samples/contact/Contact.hbm.xml"/> -->
    </session-factory>
</hibernate-configuration>
```

Create a destination that uses the annotated Java class

To use the annotated Java class, create a destination in the data-management-config.xml file named hibernate-contact that uses the HibernateAnnotationsAssembler.

Note: If the data-management-config.xml file already contains a destination named hibernate-contact, comment it out. Do not delete it because it is the destination used by the Hibernate example that uses a mapping file. For more information on that example, see “[Using Hibernate mapping files](#)” on page 281.

The following example shows an excerpt from the data-management-config.xml file that defines the hibernate-contact destination:

```
<destination id="hibernate-contact">
    <adapter ref="java-dao"/>
    <properties>
        <use-transactions>true</use-transactions>
        <source>flex.data.assemblers.HibernateAnnotationsAssembler</source>
        <scope>application</scope>
        <network>
            <paging enabled="false" pageSize="10" />
        </network>
        <item-class>flex.samples.contactannotate.Contact</item-class>
        <server>
            <fill-configuration>
                <use-query-cache>false</use-query-cache>
                <allow-hql-queries>true</allow-hql-queries>
            </fill-configuration>
        </server>
    </properties>
</destination>
```

Edit Contact.as

Edit the lcds-samples\hibernate\src>Contact.as client-side source code file to use the annotated Contact class.

Contact.as contains the following line:

```
[RemoteClass(alias="flex.samples.contact.Contact")]
```

Change the name of the class to reflect the new directory location, as the following example shows:

```
[RemoteClass(alias="flex.samples.contactannotate.Contact")]
```

After you edit Contact.as, recompile the main application file, contactmgr.mxml. For more information on setting up your development environment to compile the Test Drive examples, see “[Building and deploying LiveCycle Data Services applications](#)” on page 9.

You can now view the application by requesting the following URL:

<http://localhost:8400/lcds-samples/hibernate/index.html>

Configuring destinations to use Hibernate

Create a destination in the data-management-config.xml or services-config.xml file to reference a Java class that uses Hibernate. Typically, you use the data-management-config.xml file for all configuration related to Data Services.

The following example shows a destination that uses the HibernateAssembler. This HibernatePerson destination has a many-to-one relationship with a HibernateGroup destination.

```
<destination id="HibernatePerson" channels="rtmp-ac">
    <adapter ref="java-adapter" />
    <properties>
        <source>flex.data.assemblers.HibernateAssembler</source>
        <scope>application</scope>
        <!-- Specify any managed association elements. -->
        <metadata>
            <many-to-one property="group"
                destination="HibernateGroup" lazy="true"/>
            <one-to-many property="friends" read-only="true" paged-collection="true"
                destination="HibernatePerson" lazy="true"/>
        </metadata>
        <network>
            <paging enabled="true" pageSize="10" />
        </network>
        <server>
            <!-- If this element is not present, the adapter assumes that
                the Hibernate entity has the same name as the destination id.
            -->
            <item-class>contacts.hibernate.Person</item-class>
            <!-- Conflict modes determine whether to check for conflicts at all
                and if so whether to check on changed properties only or whether
                to verify that the client had correct data for the entire object.
                Valid values for update-conflict-mode are none, property, and object.
                Valid values for delete-conflict-mode are none and object.
            -->
            <update-conflict-mode>property</update-conflict-mode>
            <delete-conflict-mode>object</delete-conflict-mode>
            <fill-configuration>
                <use-query-cache>false</use-query-cache>
                <allow-hql-queries>true</allow-hql-queries>
            </fill-configuration>
        </server>
    </properties>
</destination>
```

The following table describes each of the destination elements that are specific to the Hibernate Assembler. These elements are children of the `server` element.

Element	Description
create-security-constraint	(Optional) Applies the security constraint referenced in the <code>ref</code> attribute to create requests.
delete-conflict-mode	Verifies that delete data from a Flex client application is not stale. If omitted, no data verification occurs. Valid values are <code>none</code> and <code>object</code> . A data conflict results in a <code>DataConflictEvent</code> on the client.
delete-security-constraint	(Optional) Applies the security constraint referenced in the <code>ref</code> attribute to delete requests.
fill-configuration	(Optional) The <code>allow-hql-queries</code> child element determines whether HQL queries can be used. The default value is <code>false</code> . The <code>use-query-cache</code> element is an option to the Hibernate query method which lets Hibernate cache queries. The default value is <code>false</code> .

Element	Description
hibernate-config-file	(Optional) Location of Hibernate configuration file; default is to look for this file in classpath.
item-class	(Optional) Exposes custom handling of type mappings on query parameter bindings; LiveCycle Data Services honors the Hibernate type mappings provided in the Hibernate configuration for a persistent Java object. To gain access to the class metadata for such programmatic query creation and parameter type mapping, you configure a destination with the name of the Hibernate entity that represents the persistent Java object. If omitted from the configuration, the Hibernate entity type is assumed to be the same as the destination id.
metadata	(Optional) Specify managed association elements such as one-to-one, many-to-one, many-to-many, and one-to-many. For more information, see “ Hierarchical data ” on page 293. You can also specify an identity element. If you include the identity element, the property attribute must specify the name of a property in the Java class. Include the identity element when you want to take complete control over the metadata section for your data type. You can then handle Hibernate associations without creating associations in Data Management configuration so that they are treated as value objects. If you omit the identity element, the metadata section is automatically generated from the Hibernate configuration.
page-queries-from-database	(Optional) If you set this element to true, the Hibernate Assembler only fetches the pages of filled collections that clients have requested from the database. By default, the assembler fetches all items in the queries from the database and pages them out to the clients one page at a time. You might have to implement a COUNT database query when you use this approach if your clients must know the size of the query before they have paged in the last items.
source	Specifies the assembler class, either HibernateAssembler or HibernateAnnotationsAssembler, which is the class that handles data changes. These elements are subelements of the properties element.
scope	(Optional) Specifies the scope of the assembler; the valid values are application, session, and request. The default value is application.
update-conflict-mode	Verifies that update data from a Flex client application is not stale. If omitted, no data verification occurs. Valid values are none, property, and object. A data conflict results in a DataConflictEvent on the client.
update-security-constraint	(Optional) Applies the security constraint referenced in the ref attribute to update requests.

Note: if you declare a Hibernate destination in your data-management-config.xml file that has a reference to entities with no destination, a destination is created automatically. Its name is the entity-name in the Hibernate configuration, which is often the class name. This destination is not accessible directly to the client through fill calls unless it is explicitly declared in the data-management-config.xml file. This prevents execution of queries against domain objects without the an explicit decision to expose those queries.

Using server-side logging with the Hibernate assemblers

The Hibernate assemblers logs messages by using the server-side logging system that is configured through the services-config.xml file. For information about server-side logging, see “[Logging](#)” on page 420. To log Hibernate assembler messages, use the following log filter:

```
<filters>
  <pattern>Service.Data.Hibernate</pattern>
</filters>
```

The Model Assembler

The Model Assembler feature lets you generate Data Management Service destinations based on an application model without writing any Java code. The generated server code uses the Hibernate object/relational persistence and query service. It extends the Hibernate Assembler.

The Model Assembler logs messages through the server-side logging system that is configured in the services-config.xml file. For information about server-side logging, see “[Logging](#)” on page 420. To log Model Assembler messages, use the Services.Data.Fiber log filter pattern or a wildcard pattern.

More Help topics

“[Model-driven applications](#)” on page 326

Hierarchical data

The Data Management Service provides several ways to manage the hierarchical relationships among managed objects. You can also mix different approaches in the same application, depending on your application requirements.

About hierarchical data

A *hierarchical collection* is hierarchical data that includes at least one complex object whose properties are not simple primitive values. For example, consider a collection that contains a Person object that has an Address property, which itself has street, city, and state properties.

There are three techniques for managing hierarchical data with the Data Management Service:

Hierarchical values approach A single Data Management Service destination manages an entire tree of data. In this approach, you treat the complex property as a value object.

Managed associations approach You define a parent destination that has declared associations to child destinations, which manage the values of the properties of the child objects. In this approach, the complex property references other entities.

Note: You use the managed associations approach in model-driven development with the Model Assembler, but you do not manually write Java code or configure destinations. Instead, you create associations in a data model and deploy the model to the server. For more information, see “[Model-driven applications](#)” on page 326.

Query approach You implement a relationship between two objects using queries where one of the parameters of a fill method defined in an assembler is the property defined as the identity property of the related object.

You can use one of these approaches, or you can combine approaches on a property-by-property basis.

In the hierarchical values approach and the managed associations approach, the parent destination returns the same graph of objects from its fill invocation. However, in the hierarchical values approach, the adapter of the parent object is responsible for saving the state of the entire complex property. The parent is also responsible for updating the entire graph in response to a sync invocation; on any change, it receives the complete new graph for the object and updates that graph with any changes.

For the managed associations approach, the parent destination manages the identity properties of the referenced child objects and the relationship between the parent and child objects. The child assembler owns the state of the child objects. A side benefit of using the managed association approach is that you can use destinations independently of each other in one application, while using them together in another application.

In the managed associations approach, the configuration for the parent destination contains an additional element that defines an association for the child object that has its own destination. This association tag refers to the separate destination that is responsible for managing the state of the child object.

When you use the hierarchical values approach or the managed association approach, you either mark each class in the top-level object graph of managed classes with the [Managed] metadata tag or explicitly implement the mx.data.IManaged interface. This ensures that the managed objects support the proper change events and that getter methods for one-to-one association properties that are lazily loaded can throw ItemPendingErrors. For more information about lazy loading and ItemPendingErrors, see “[The managed association approach](#)” on page 295.

The following example shows the code for an ActionScript class that uses the [Managed] metadata tag. This Employee class is used in an ArrayCollection of Employees, which is used as a property of a Company class.

Note: You do not manually code managed ActionScript classes, such as the following class, when you use model-driven development. Instead, ActionScript classes are generated automatically from a model file. For more information, see “[Model-driven applications](#)” on page 326 and the [Application Modeling Technology Reference](#).

```
package samples.crm
{
    [Managed]
    [RemoteClass(alias="samples.crm.Employee")]
    public class Employee {
        public var employeeId:int;
        public var firstName:String = "";
        public var lastName:String = "";
        public var title:String = "";
        public var phone:String = "";
        public var email:String = "";
    }
}
```

The following example shows a class that implements the mx.data.IManaged interface instead of using the [Managed] metadata tag. When you implement mx.data.IManaged, you must support the mx.core.IUID interface, which requires a uid property. You also must include the referenceIds and destination variables declared under the last comment block in this class.

```
import mx.core.mx_internal;
import mx.data.Managed;
import mx.data.IManaged;
import mx.utils.UIDUtil;

[RemoteClass(alias="foo.bar.Customer")]
public class Customer implements IManaged {
    public function Customer() {
        super();
    }

    [Bindable(event="propertyChange")]
    public function get firstName():String {
        _firstName = Managed.getProperty(this, "firstName", _firstName);
        return _firstName;
    }

    public function set firstName(value:String):void {
        var oldValue:String = this._firstName;
        _firstName = value;
        Managed.setProperty(this, "firstName", oldValue, _firstName);
    }
}
```

```
}

// all implementors of IManaged must support IUID which requires a uid
// property
[Bindable(event="propertyChange")]
public function get uid():String {
    // If the uid hasn't been assigned a value, just create a new one.
    if (_uid == null) {
        _uid = UIDUtil.createUID();
    }
    return _uid;
}

public function set uid(value:String):void {
    _uid = value;
}

// These are special requirements of any object that wants to be
// "managed" by a DataService.
// The referencedIds is a list of object IDs that belong to properties that
// are lazily loaded.
// The destination is used to look up metadata information about the
// associations this object has been configured with on the server.
mx_internal var referencedIds:Object = {};
mx_internal var destination:String;

private var _firstName:String;
private var _uid:String;
}
```

Note: The referencedIds and destination variables in this example are in the mx_internal namespace, which Adobe uses to mark things that may change in future versions of the Flex framework. Your inherited class or component may not work with future releases of Flex.

The managed association approach

In the managed association approach, the parent destination is not responsible for processing updates for changes made to properties of the child objects. It is only responsible for processing the changes to the identity property values of children referenced by the parent. Suppose a Person object has an Address property. Using the hierarchical values approach, the Person destination would handle a change to the city of a person address. However, in the managed association approach, the Address destination would process a change to the city.

In the managed association approach, the Person destination would process a change if someone assigned an Address with a different identity property to the Address property of the Person object. You can also configure whether the relationship supports lazy loading of object graphs from the server to the client.

Note: You use the managed association approach with model-driven development. The Java code and configuration file examples included here do not apply to model-driven development. For more information about using associations with model-driven development, see “[Model-driven applications](#)” on page 326 and the [Application Modeling Technology Reference](#).

The destination relationships you configure can be unidirectional or bidirectional. When you configure destinations in a configuration file, unidirectional relationships use the many-to-one, one-to-one, and one-to-many elements, and bidirectional relationships use the many-to-many element. These *managed association elements* are subelements of the metadata element, which is a supplement of the properties element.

Note: Managing a one-to-many relationship can require much more computation than managing a many-to-one relationship.

Some managed associations rely on database tables. To establish a many-to-many relationship between these tables, create a join table rather than using a foreign key.

You set the `read-only` attribute of an association property to true for the inverse side of a two-way relationship. This setting indicates that the assembler does not depend on the state of this property being populated, which ensures that dependent items are created before independent items. Setting the `read-only` attribute to `true` does not mean that you cannot edit the association value. Rather, it means that the association is read only with respect to the assembler, which does not use that value to update the database. Instead, the assembler uses the other side of the relationship.

Managed association example

The Sales Builder sample application provides an example of a many-to-one relationship. On the Flex client side of the application, a `DataService` component named `account` manages a collection of `Account` objects that works with a destination named `account` on the server. The `Account` objects have a set of properties, including a `salesRep` property of type `SalesRep`, which works with a destination named `sales-rep` on the server. The `account` destination defines a many-to-one relationship to the `sales-rep` destination, so `SalesRep` changes, additions, or deletions are propagated up to the corresponding `Account` object or objects to which the particular `SalesRep` objects belong.

The following example shows the source code for the `Account` class:

```
package com.salesbuilder.model
{
    [Managed]
    [RemoteClass(alias="com.salesbuilder.model.Account")]
    public class Account
    {
        public var accountId:int;
        public var name:String;
        public var phone:String;
        public var fax:String;
        public var ticker:String;
        public var ownership:String;
        public var numberEmployees:int;
        public var annualRevenue:Number = 0;
        public var priority:int;
        public var address1:String;
        public var address2:String;
        public var city:String;
        public var zip:String;
        public var notes:String;
        public var url:String;
        public var rating:int;
        public var currentYearResults:Number = 0;
        public var lastYearResults:Number = 0;
        public var industry:Industry;
        public var salesRep:SalesRep;
        public var category:AccountCategory;
        public var state:State;
    }
}
```

The following example shows the source code for the `SalesRep` class:

```
package com.salesbuilder.model
{
    [Managed]
    [RemoteClass(alias="com.salesbuilder.model.SalesRep")]
    public class SalesRep
    {
        public var salesRepId:int;
        public var firstName:String;
        public var lastName:String;

        public function get fullName():String
        {
            return firstName + " " + lastName;
        }
    }
}
```

On the server side, Java-based Account and SalesRep objects represent accounts and sales representatives. These classes are mapped to the corresponding client-side Account and SalesRep objects.

Destination configuration

The following example shows relevant sections of the account and sales-rep destinations, including the `manytoone--` element in the account destination that defines the relationship between the destinations. The boldface text highlights the configuration for the many-to-one relationship.

```
...
<!-- account destination has a many-to-one relationship to the sales-rep destination -->
<destination id="account">
    <properties>
        <source>com.salesbuilder.assembler.AccountAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="accountId" undefined-value="0"/>
            <many-to-one property="salesRep" destination="sales-rep" lazy="true"/>
            <many-to-one property="industry" destination="industry" lazy="true"/>
            <many-to-one property="category" destination="account-category" lazy="true"/>
            <many-to-one property="state" destination="state" lazy="true"/>
        </metadata>
        <network>
            <paging enabled="false" pageSize="10" />
        </network>
    </properties>
</destination>

<!-- sales-rep destination -->
<destination id="sales-rep">
    <properties>
        <source>com.salesbuilder.assembler.SalesRepAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="salesRepId" undefined-value="0"/>
        </metadata>
        <network>
            <paging enabled="false" pageSize="10" />
        </network>
    </properties>
</destination>
```

The query approach

You can implement a relationship between two objects by using queries where one of the parameters of a `fill()` method in the assembler is the identity property (defined in a destination definition in the `data-management-config.xml` file) of the related object. The query approach is more efficient for large collections of objects than the managed association approach. For example, it would make sense to use it if you have a Company object that has a large number of Employee instances.

The following code from the CRM sample application shows the `crm-company` destination definition, which declares the `companyId` property as the identity property:

```
<destination id="crm-company">
    <adapter ref="java-dao" />
    <properties>
        <source>flex.samples.crm.company.CompanyAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="companyId"/>
        </metadata>
        ...
    </properties>
</destination>
```

The following example shows the `fill()` method of the `EmployeeAssembler` class in the CRM application. The boldface text highlights the part of the method that finds employees by company, based on the numeric `company.companyId` property value provided in a client's fill request.

```
...
public Collection fill(List fillParameters) {
    if (fillParameters.size() == 0){
        return dao.getEmployees();
    }
    String queryName = (String) fillParameters.get(0);
    if (queryName.equals("by-name"))
        return dao.findEmployeesByName((String) fillParameters.get(1));
if (queryName.equals("by-company"))
    return dao.findEmployeesByCompany((Integer) fillParameters.get(1));
    return super.fill(fillParameters); // throws a nice error
}
```

The boldface code in the following example is the corresponding client-side fill request that gets employees by company:

```
private function companyChange():void {
    if (dg.selectedIndex > -1)
    {
        if (company != companies.getItemAt(dg.selectedIndex))
        {
            company = Company(companies.getItemAt(dg.selectedIndex));
            dsEmployee.fill(employees, "byCompany",
                company.companyId);
        }
    }
}
```

The query approach to implementing a relationship has some characteristics that make it not appropriate for all situations. When you change an employee `companyId`, the Data Management Service must update the results of the fill methods affected by that change for clients that display the list of employees to be updated automatically. By default, it uses the auto-refresh feature to re-execute every outstanding fill method that returns employees. You can adjust this behavior to avoid re-executing these queries so aggressively, but that requires some additional coding in your assembler.

The query approach also does not detect conflicts when you update that association. You would not want to detect a conflict if two clients added an employee to the same company at roughly the same time, but you could want to detect a conflict if two clients simultaneously update addresses for the same customer. With managed association properties, you can use the Data Management Service conflict detection mechanism to detect data conflicts on the complete contents of the collections; however, with a fill method, conflicts are only detected on the properties of the item, not the filled collection the item was returned in.

The CRM application implements a relationship between companies and employees in the `fill()` method of the `EmployeeAssembler` class. The source code for the Java assembler and DAO classes are in the `WEB_INF/src/flex/samples/crm` directory of the `lcds-samples` web application.

Managing class hierarchies

If you have strongly typed Java classes that extend each other and you want to manage both the supertype and the subtype, there are two strategies you can use. You can use the *single destination approach* in which one destination manages the entire type hierarchy, or you can use the *multiple destinations approach* in which you define a separate destination for each managed class in the hierarchy. You cannot mix these two approaches for the same class hierarchy.

You can only use the single destination approach if all of the classes in the class hierarchy have the same association properties. If a subclass introduces a new association, you define a separate destination for each concrete class in the class hierarchy. In both strategies, the identity properties must be the same for all managed classes in the type hierarchy. Additionally, in both approaches there must be ActionScript classes defined for each concrete Java class.

For the multiple destinations approach, you also specify the `item-class` element for each destination to refer to the entity class name of the object associated with the destination. You should also specify the `extends` attribute of the `metadata` element to refer to the super type destination. When using the multiple destinations approach, the following rules apply:

- The `fill` or `getItem` method can return instances of the class defined for this association (if any) or any subclasses of that destination.
- The `updateItem`, `createItem`, or `deleteItem` methods are called on the assembler defined for the most specific destination defined for the given class.
- Subclasses automatically inherit identity properties and associations and do not have to be redeclared.
- If you do not define a separate assembler instance (using a `source`, or `factory` element), the assembler instance for the `extends` destination is used for the child destination.

When you use the `extends` attribute of the `metadata` element in a destination, you use the `item-class` element to bind each destination to a specific Java class; for information about the `item-class` element, see “[Data Management Service clients](#)” on page 226. You also must have an ActionScript class mapped to each Java class so the type is preserved as objects are sent to the client.

The instances returned by your assembler from a given destination should be instances managed by this destination or the destination of a subtype. An extended destination inherits the identity properties and associations of its base type. It can extend more than one destination, but those destinations must have the same identity properties.

If you want to use one assembler instance for an entire class hierarchy, make sure that you use the `attribute-id` element so that each destination uses the same attribute name to store the class and also ensure the source and scope properties are copied among the destinations.

Single destination approach for managing class hierarchies

The following examples show the source code of Java and ActionScript classes and destination configuration for the single destination approach.

Foo.java

```
package samples.oneDestExtends;

public class Foo
{
    public Foo() {}

    public int id;

    public int property1;
    public int property2;

    public Foo selfReference; // my store ref to instance of Foo or Bar
}
```

Bar.java

```
public class Bar extends Foo
{
    public int property3;
    public int property4;

    // NOTE: cannot add additional managed associations in the subclass using this approach
}
```

Foo.as

```
[Managed]
[RemoteClass(alias="samples.oneDestExtends.Foo")]
public class Foo implements java.io.Serializable
{
    public function Foo() {}

    public var id:int
    public var property1:int
    public var property2:int

    public var selfReference:Foo;
}
```

Bar.as

```
[Managed]
[RemoteClass(alias="samples.oneDestExtends.Bar")]
public class Bar extends Foo
{
    public function Bar()
    {
        super();
    }

    public var property3:int;
    public var property4:int;
}
```

data-management-config.xml

```
...
<destination id="oneDestExtends.foo">
    <properties>
        <metadata>
            <identity property="id"/>
            <many-to-one property="selfReference" destination="oneDestExtends.foo"/>
        </metadata>
        <source>oneDestExtends.FooAssembler</source>
    </properties>
</destination>
...

<!-- No destination required for Bar in this approach -->
```

Multiple destination approach for managing class hierarchies

The following example code shows the Java class source code and destination configuration for an application that uses the multiple destinations approach. In this case, a Node class has a ParentNode subclass. The Node class has an association called `parent` of type ParentNode and the ParentNode class adds an association called `children` of type Node.

Node.java:

```
package samples.multiDestExtends;

public class Node implements java.io.Serializable
{
    public Node() {}

    public int id;

    public ParentNode parent;
}
```

ParentNode.java:

```
public class ParentNode extends Node
{
    // Contains either Node or ParentNode instances
    public List children = new LinkedHashSet();
}
```

data-management-config.xml

```
...
<destination id="multiDestExtends.Node">
    <properties>
        <metadata>
            <identity property="id" undefined-value="-1"/>
            <many-to-one property="parent"
                destination="multiDestExtends.ParentNode" lazy="true"/>
        </metadata>

        <item-class>samples.multiDestExtends.Node</item-class>
        <source>samples.multiDestExtends.NodeAssembler</source>
        <scope>application</scope>
    </properties>
</destination>

<destination id="multiDestExtends.ParentNode">
    <properties>
        <metadata extends="multiDestExtends.Node">
            <one-to-many property="children"
                destination="multiDestExtends.Node" read-only="true"
                lazy="true" page-size="2"/>
        </metadata>
        <item-class>samples.multiDestExtends.ParentNode</item-class>

        <!-- inherits same assembler instance by default -->
        </properties>
</destination>
```

Node.as:

```
[RemoteClass(alias="samples.treepagingjdbc.Node")]
[Managed]
public class Node
{
    public var id : int;
    public var name:String;
    public var parent:ParentNode;

    public function Node()
    {
        super();
    }
}
```

ParentNode.as

```
import mx.collections.ArrayCollection;

[RemoteClass(alias="samples.treepagingjdbc.ParentNode")]
[Managed]
public class ParentNode extends Node
{
    public var children:ArrayCollection;

    public function ParentNode()
    {
        super();
    }
}
```

Data paging

Paging can drastically improve performance of applications by decreasing query times and reducing the amount of consumer memory. LiveCycle Data Services supports three types of paging: client-to-server paging, server-to-data-source paging, and association paging.

With client-to-server paging enabled, the server reads a collection of objects in its entirety into server memory; the data is then paged on demand to client applications. Server-to-data-source paging extends the benefits of paging to the application data source so that pages are only read into server memory as they are needed. Association paging allows you to bring associated objects to the client on demand.

Client-to-server paging

When client-to-server paging is enabled, the initial fill request from a client causes the assembler associated with a Data Management Service destination to retrieve the entire collection of objects. The server then sends the first page of items to the Flex client. As the client code tries to access ArrayCollection elements that are not resident, additional pages are retrieved from the server.

To enable client-to-server paging when you are not using model-driven development, add a `paging` element to the network properties section of your destination definition in the `data-management-config.xml` file. You can set the default page size in the `page-size` attribute of the `paging` element.

The following example shows a destination definition with client-to-server paging enabled:

```
<destination id="inventory">
    <properties>
        <source>flex.samples.product.ProductAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="productId"/>
        </metadata>
        <network>
            <paging enabled="true" pageSize="10" />
        </network>
    </properties>
</destination>
```

To enable client-to-server paging and set the page size for model-driven development, set the `paging-enabled` and the `page-size` annotations on an entity , as the following example shows:

```
<entity name="Book" persistent="true">
    <annotation name="DMS">
        <item name="paging-enabled">
            true
        </item>
        <item name="page-size">
            10
        </item>
    </annotation>
    ...
</entity>
```

Server-to-data-source paging

When the original fill query is resource-intensive or returns more results than are desirable to cache on the server, you can extend paging from the Flex client all the way to the data source. In this case, the assembler retrieves items from the data source one page at a time, and only when the client requests a page. This type of paging is called server-to-data-source paging. This type of paging is only enabled if client-to-server paging is enabled.

When you use server-to-data-source paging, each time the client requests a page, the assembler is asked for that page of items and the items are sent directly to the client. This type of paging is supported whether or not you set the `autoSyncEnabled` property of the `DataService` component on the client to `true`. This configuration is partially supported for the `SQLAssembler`. It is supported for the `HibernateAssembler`, and you can implement it with your own Java Assembler implementation.

To enable server-to-data-source paging for the `Hibernate Assembler`, set the `page-queries-from-database` attribute to `true` in the server properties of the destination. To enable server-to data source paging with the `SQL Assembler`, set the `page-query` element to `true`. The `page-query` element is a child of the `fill` element.

The steps required to enable server-to-data-source paging for a custom assembler depend on whether the assembler implements the `flex.data.assemblersAssembler` interface. If your Assembler does implement this interface, you should define the `useFillPage(List fillParameters)` method to return `true` for the fill parameters where you want to extend paging to the data source. You then implement the `fill(List fillParameters, int startIndex, int numberOfRows)` method to fetch subsets of your collection from your data source based on the `startIndex` and `numberOfRows` parameters. The Data Management Service invokes this method with parameters to the pages that clients request.

If your assembler does not implement the `flex.data.assemblersAssembler` interface, set the `custom` attribute of the `paging` element to `true` in the network properties section of the destination definition in the `data-management-config.xml` file. Next, specify a `fill` method that matches the `fill` parameters used by your client with two additional parameters: the start index and the number of items requested.

When you use server-to-data-source paging, your assembler must implement the `count()` method. For more information, see “[Data paging](#)” on page 303.

When you enable paging for model-driven development, server-to-data-source paging is enabled by default. To disable server-to-data-source paging for model-driven development, set the `page-queries-from-database` annotation on an entity filter to `false`, as the following example shows:

```
<entity name="Book" persistent="true">
    ...
    <filter>
        <annotation name="DMS">
            <item name="page-queries-from-database">
                false
            </item>
        </annotation>
    ...
</filter>
...
</entity>
```

Dynamic sizing

When server-to-data-source paging is enabled, the server does not immediately know the total size of the collection that is being paged because it is retrieving the collection one page at a time. To return the size of a data-source-paged collection to the client, the Data Management Service invokes the `count()` method of the assembler after the client makes its initial fill request. However, the `count` query is often almost as expensive as its corresponding `select` query. The dynamic sizing option lets you avoid potentially expensive `count` queries.

To enable dynamic sizing when you are not using model-driven development, implement your `count()` method to return `-1` for appropriate fill parameters. With dynamic sizing, the paged fill method of the assembler is called with a `startIndex` value of `0` and the number of items is set to the `pageSize + 1`. If the assembler method returns less than the number requested, the size of the fill is known. If it returns the `pageSize+1` items requested, `pageSize` items are returned to the client but the client sets the collection size to `pageSize + 1` with one empty slot at the end. When the client requests that empty item, the next `pageSize+1` items are requested and the process repeats until the assembler returns less than `pageSize+1` items.

If you are using the Hibernate assembler and the HQL query sent from the client is a simple query, the Hibernate assembler attempts to implement a `count` query by modifying the query sent from the client. If you are using a named query, the Hibernate assembler looks for a query named `original-query-name.count`. If that query exists, it uses it to compute the size of the paged collection. Otherwise, it uses the dynamic sizing approach.

To enable dynamic sizing for model-driven development, set the `dynamic-sizing` annotation on an entity to `true`, as the following example shows:

```
<entity name="Book" persistent="true">
    <item name="dynamic-sizing">
        true
    </item>
...
</entity>
```

Association property loading

The following options are available to control how the client loads association properties:

- Loading only the identity properties
- Not loading associated items until they are used
- Paging associated items
- Using paged updates

By default, the entire associated collection is sent to the client when the parent item is fetched. For general information about association properties, see “[Hierarchical data](#)” on page 293.

Loading only the identity properties

To avoid loading entire associated items when the parent object is loaded, the Data Management Service provides the lazy load option. With lazy loading, only the identity properties of the associated items are loaded with the parent object. The associated items are passed to the client by reference (identity property) instead of by value (entire object). When the client attempts to access an associated item for which it only has an identity property, an `ItemPendingError` is thrown. For information about `ItemPendingErrors`, see “[Item pending errors](#)” on page 308.

When using lazy loading, the assembler only returns the identity properties of the referenced objects. It does not have to fully populate these referenced objects.

If you are not using model-driven development, enable lazy loading by setting the `lazy` attribute of an association element, such as a `many-to-one`, `one-to-one`, `one-to-many`, or `many-to-many` element, to `true`.

For model-driven development, enable lazy loading by setting the `lazy` annotation on an entity property to `true`, as the following example shows:

```
<entity name="Book" persistent="true">
    <property name="name" type="string">
        <annotation name="DMS">
            <item name="lazy">true</item>
        </annotation>
        ...
    </property>
</entity>
```

Not loading associated items until they are needed

When lazy loading is enabled, the assembler only has to fetch associated identity properties to satisfy a fill request. Although using lazy loading lets you avoid retrieving entire associated objects, it can still be inefficient for large association collections. To avoid loading an associated collection until it is needed on the client, the Data Management Service provides the load-on-demand option.

When you set the `load-on-demand` to `true` on an association, the association property value is ignored from the initial `getItem()` and `fill()` method calls. When the client first accesses the property, the `getItem()` method is called again for the parent item, and this time the property value is fetched. If `load-on-demand` is enabled and `lazy` is set to `true` for the association, when the associated property is first accessed only the identity properties of the items are fetched and the `getItem()` method is called on each identity property as that item is needed. If `load-on-demand` is enabled and `lazy` is set to `false`, the entire items are fetched when the association property is accessed.

For model-driven development, enable load-on-demand by setting the `load-on-demand` annotation on an entity to `true`, as the following example shows:

```
<entity name="Book" persistent="true">
    <annotation name="DMS">
        <item name="load-on-demand">
            true
        </item>
    </annotation>
    ...
</entity>
```

Paging associated items

For multi-valued association properties (for example, one-to-many and many-to-many properties), consider bringing in associated items or their identity properties one page at a time, instead of the entire associated collection when it is needed as load-on-demand lets you do. As with load-on-demand, the initial `getItem()` and `fill()` method calls do not return any of the association property values to the client. When the property is accessed on the client, the `getItem()` method is called again for the parent item. This time it must return a page of the associated collection. The length is returned to the client and the page containing the requested item is also returned.

Like load-on-demand, association paging is set independently of the `lazy` property. If the `lazy` property is set to `true` and either load-on-demand or paging is used, when the client attempts to access an item for which only the identity property is available, another `ItemPendingError` is thrown and the item is retrieved in its entirety by invoking the `getItem()` method.

To enable association paging between the client and the server when not using model-driven development, set the `page-size` attribute of an association element in a destination to the desired value, as the following example shows:

```
<one-to-many property="members" destination="Person" page-size="3"/>
```

To enable server-to-data-source association paging when not using model-driven development, in addition to the `page-size` attribute, set the `paged-collection` attribute of an association element in a destination to the desired value as the following example shows. This setting lets you retrieve a subset of the collection from the assembler.

```
<one-to-many property="members" destination="Person" page-size="3" paged-collection="true"/>
```

For model-driven development, enable association paging by setting the `page-size` annotation of an association property of an entity to `true`, as the following example shows:

```
<entity name="Company" persistent="true">
    <property name="employee" type="Employee">
        <annotation name="DMS">
            <item name="page-size">5</item>
        </annotation>
    ...
</property>
</entity>
```

For model-driven development, server-to-data-source association paging is enabled by default when you set the `page-size` annotation. This is because, unlike custom assemblers, the Model Assembler is guaranteed to implement the `useFillPage(List fillParameters)` method to return `true` for its fill parameters.

Using paged updates

The `paged-updates` Boolean attribute of an association property controls how changes are propagated between clients and the server. When the `paged-updates` attribute is set to `false` (default value), the entire collection is sent when the association value changes. When the `paged-updates` attribute is set to `true`, only the identity properties of the newly added or removed items are sent.

If the client uses paged updates to update a collection property, when those changes are committed to the server, the `updateCollectionProperty()` method on the Assembler interface is invoked to update the collection property.

When association values are paged (`page-size` value is set to a value greater than zero), paged updates are automatically enabled. However, enabling load-on-demand for associations does not automatically enable paged updates.

For model-driven development, enable paged updates by setting the `paged-updates` annotation of an entity property to `true`, as the following example shows:

```
<entity name="Company" persistent="true">
<property name="name" type="string">
    <annotation name="DMS">
        <item name="paged-updates">
            false
        </item>
    </annotation>
    ...
</property>
</entity>
```

Paging and property specifiers

The `PropertySpecifier` object tells assembler operations, such as `getItem()`, `fill()`, and `refreshItem()` methods, which properties of a data item to populate. When a client discovers that it has the identity property of an item locally but not the entire item, it invokes a `getItem()` method on the server using the default property specifier. The default property specifier denotes that load-on-demand and paged association properties are not required to be populated on the item returned by the assembler for the `getItem()` call. As a result, you can code your custom assembler to skip such properties in its implementation of the `getItem(Map identity, PropertySpecifier ps)` method when `ps` is the default property specifier. Alternatively, your implementation of this method can invoke the Boolean `PropertySpecifier.includeProperty()` method with a specific property name to determine whether the property must be populated on this invocation of the `getItem()` method. You can use this method so that your assembler behavior is not hard-coded to know the settings of the `load-on-demand` and `page-size` properties that you use for a specific destination. For example, the implementation of the Hibernate assembler uses this variant to determine which properties must be fetched, and it reacts to the configuration changes that you make without requiring code changes to the assembler.

Item pending errors

When the Flex client accesses a lazily loaded value for the first time, an `ItemPendingError` is thrown either from the `ArrayCollection.getItem()` method for multiple-valued associations or from the `Managed.getProperty()` method for single-valued associations. This causes a request to fetch the referenced object, and an event handler in the `ItemPendingError` is invoked to notify the client when the item is available. When you use Flex data binding, this behavior happens automatically when you set the `lazy` attribute to `true`; the data appears when it is needed. If you do not use data binding in conjunction with lazy loading, you can catch the `ItemPendingError` in your code and handle it accordingly.

An `ItemPendingError` is thrown when retrieving an item from an `ArrayCollection` requires an asynchronous call. When you bind a managed `ArrayCollection` to a List-based control other than a Tree control, `ItemPendingErrors` are handled automatically when you have the `paging` element set to `true` on the destination. If the receiver of this error needs notification when the asynchronous call completes, it must use the `addResponder()` method and specify an object that supports the `mx.rpc.IResponder` interface to respond when the item is available. The `mx.collections.ItemResponder` class implements the `IResponder` interface and supports a `data` property.

In the following example, an ItemPendingError is thrown and caught when a group object is not yet available to a groupCollection object on the client. The groupCollection object is filled from the AssocGroup destination on the server. The destination has an admin property that is a lazily loaded reference to an AssocPerson instance. The printAdminName() method is called for the first group in the groupCollection object. If a group is not available, it throws an ItemPendingError. An ItemResponder is registered to handle the ItemPendingError; the ItemResponder calls the printAdminName() method when the group is available. If there is a fault, the ItemResponder calls the fetchAdminError() method.

```
...
import mx.collections.ItemResponder;
import mx.messaging.messages.ErrorMessage;
import mx.collections.errors.ItemPendingError;
...
public function printAdminName (data:Object, group:Object):void {
    trace(group.admin.firstName);
}
public function fetchAdminError(message:ErrorMessage):void {
    trace("error occurred fetching admin: " + message.faultString);
}
// This method is called to retrieve a "group" object
// that has a lazily loaded reference to a Person in its admin property.
// The printAdminName function may throw the ItemPendingError
// when it retrieves the group.admin property if that object has
// not yet been loaded by the client.
// The function registers a listener that calls printAdminName
// again when the value is available.
public function dumpAdminName():void {
    try {
        printAdminName(null, groupCollection.getItemAt(0));
    }
    catch (ipe:ItemPendingError) {
        trace("item pending error fetching admin.");
        ipe.addResponder(new ItemResponder(printAdminName, fetchAdminError,
            groupCollection.getItemAt(0)));
    }
}
...

```

Hibernate association paging configuration

The following configuration example shows a Hibernate destination with a one-to-many element that has the pagesize- attribute set to a nonzero value:

```
<destination id="account.hibernate">
    <adapter ref="java-dao" />
    <properties>
        <use-transactions>true</use-transactions>
        <source>flex.data.assemblers.HibernateAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="id"/>
            <one-to-many property="accountContacts" destination="accountContact.hibernate"
                read-only="true" pageSize="2" />
            <many-to-one property="consultant" destination="consultant.hibernate"
                lazy="true" />
        </metadata>
        <server>
            <hibernate-config-file>support-hibernate.cfg.xml</hibernate-config-file>
            <hibernate-entity>support.Account</hibernate-entity>
            <fill-configuration>
                <use-query-cache>false</use-query-cache>
                <allow-hql-queries>true</allow-hql-queries>
            </fill-configuration>
        </server>
    </properties>
</destination>
```

Occasionally connected clients

Occasionally connected data management clients work with managed data in both online and offline states. You can write an AIR-based (desktop) or browser-based clients that work online and offline. The offline cache stores the managed items, destination metadata, executed fill membership, and the uncommitted messages along with conflicts. When the application goes offline, the application maintains access to the data.

The client stores the contents of executed data management operations to disk for later retrieval when an application resumes in online mode. The offline mode lets you create an application that can work when the host is shut down, a network connection is unavailable, or the host is restarted with the application loaded.

There are three basic types of occasionally connected clients. From the most powerful to the least, the types of clients are as follows:

- AIR-based client that uses a custom offline adapter
- AIR-based client with no custom offline adapter
- Browser-based (Flash Player) client that stores data in local shared objects (LSOs)

AIR-based clients

An AIR-based client with a custom offline adapter provides the best offline experience. This type of client works with the AIR SQLite database to cache data and perform offline fill, commit, and delete operations when the client is disconnected from the LiveCycle Data Services server.

In the custom offline adapter, you write logic for SQL queries that retrieve locally cached items just as the Data Management Service assembler retrieves items from the data source on the server. Unlike the other types of occasionally connected clients, this type gives you full query capabilities; you can perform offline fill operations on locally cached data whether or not the same fill operations have already been performed while online and cached locally. The local data store is not just a cache of data that the client has already requested, but is a true local data store that is actively populated with data from the server.

Clients that do not use custom offline adapters only cache data that the user has already accessed while connected to the server. That works well for providing the same data back to the user for the same queries when the application goes offline, but not for performing offline queries that have not yet been cached.

When you do not use an offline adapter for an AIR-based client, data is stored in the AIR SQLite database in binary large object (BLOB) format. This type of client can only perform operations that have already been performed online and cached locally.

Note: When using a SQLite offline adapter, you can use an encrypted SQLite database to ensure that data is unavailable to all other users and applications running on the same machine. To encrypt the generated SQLite database, set the `DataService.encryptLocalCache` property to `true`. The database is encrypted with a random encryption key and password using the SQLite encryption feature. The password is stored locally using AIR EncryptedLocalStore support, associated with the database name.

Browser-based clients

In the case of browser-based clients, local data is stored in local shared objects (LSOs) in BLOB format. Similar to an AIR-based client without a custom offline adapter, this type of client can only perform operations that have already been performed online and cached locally.

Another limitation of this type of client is that the LSO database does not support concurrent writes. This results in an error when you try to save data to the LSO database when a save is already in progress.

Working with data offline

For all occasionally connected clients, you use a common set of APIs to create a local data store, cache data locally, connect to the server destination, and inspect the data cache.

For an AIR-based client that uses a custom offline adapter, you also create an offline adapter and set the `DataService.offlineAdapter` property in your client application code. For more information, see “[Using an offline adapter with an AIR-based client](#)” on page 317.

Creating a local data store

Set the `DataService.cacheID` property to store data locally. The `cacheID` property is a string that provides an identifier for locally stored data.

If two clients use different `cacheID` values, they use independent cached stores of data. If they use the same value for the `cacheID` property, they share the same store of data. In general, it is not a good practice to run two clients at the same time using the same `cacheID` value. A `cacheID` value of `null` or an empty string is considered unset. If you do not set the `cacheID` property, all cache methods and properties are considered inconsistent and throw errors.

You can change the value of the `cacheID` property during an application's operation. When changed, the current state of the data is flushed to local storage. All currently loaded data remains in memory, letting you selectively add or remove cached data stored on disk using the new identifier. This results in copy-on-write behavior if you set the `DataService.autoSaveCache` property to `true`.

Call the `DataService.getCacheIDs()` method to retrieve all `cacheID` values used for a single application. When two or more AIR applications run under the same domain and use the same `cacheID` value, changes are reflected in all instances as if they are the same application. This mechanism bypasses conflict detection between the running instances; there can be no conflicts between applications for any changes made locally.

The following `creationComplete()` event handler for a DataGrid control sets the `cacheID` property.

Note: For an AIR-based client that uses a custom offline adapter, you would set the `DataService.offlineAdapter` property in addition to the properties shown here. For more information, see “[Using an offline adapter with an AIR-based client](#)” on page 317.

```
< mx:Script>
    <! [CDATA [
        ...
        protected function
            dataGrid creationCompleteHandler(event:FlexEvent) :void
        {
            // Create a DataService instance.
            var myDataService:DataService = new DataService("destination1");
            // Set the unique ID for the local data store.
            myDataService.cacheID = "cache1";
            // Save the cache automatically when online.
            myDataService.autoSaveCache = true;
            // If working offline don't attempt to connect to the server.
            myDataService.autoConnect = false;
            // Query against the local store when not connected.
            myDataService.fallBackToLocalFill=true;
            // Perform a fill operation.
            myDataService.fill(users, "notorious");
        }
    ]]>
</mx:Script>
```

Caching data locally

Set the `DataService.autoSaveCache` property to `true` to automatically cache data locally. The default value is `false`. Set `autoSaveCache` to `true` to save data locally after any change, including creates, reads, updates, deletes, page requests, and lazy loading. For information about lazy loading, see “[The managed association approach](#)” on page 295.

The following `creationComplete()` event handler for a DataGrid control sets the `cacheID` property.

Note: For an AIR-based client that uses a custom offline adapter, you would set the `DataService.offlineAdapter` property in addition to the properties shown here. For more information, see “[Using an offline adapter with an AIR-based client](#)” on page 317.

```
< mx:Script>
    <![CDATA [
        ...
        protected function
            dataGrid creationCompleteHandler(event:FlexEvent):void
        {
            // Create a DataService instance.
            var myDataService:DataService = new DataService("destination1");
            // Set the unique ID for the local data store.
            myDataService.cacheID = "cache1";
            // Save the cache automatically when online.
            myDataService.autoSaveCache = true;
            // If working offline don't attempt to connect to the server.
            myDataService.autoConnect = false;
            // Query against the local store when not connected.
            myDataService.fallBackToLocalFill=true;
            // Perform a fill operation.
            myDataService.fill(users, "notorious");
        }
    ]]>
</mx:Script>
```

Fill parameters identify the items in a fill request. In the preceding example, the key for the list of items returned in the fill request is `notorious`. When the application is reloaded in a disconnected state, it requests the items with the key value of `"notorious"` from the local store.

If you set the `autoSaveCache` property to `false`, you can call the `DataService.saveCache()` method to manually save data to the local file system. The `saveCache()` method lets you control the timing of save operations. Use it to force a save of the current state of the cache to local storage. You can specify an optional parameter to provide further granularity over what is stored locally; the parameter must be a managed ArrayCollection or a managed item.

Call the `DataService.clearCache()` method to remove all locally stored data associated with the `DataService` instance. The `clearCache()` method takes an optional parameter to remove a specific piece of data from the local cache.

The following example shows the `DataService saveCache()` and `clearCache()` methods with an `ArrayCollection` as a parameter:

```
<mx:Application creationComplete="loadData()">
    <mx:DataService id="usersOfD" destination="D" />
    <mx:DataGrid id="usersGrid" dataProvider="{users}">
    </mx:DataGrid>
    <mx:Button label="Save"
        click="usersOfD.saveCache(ArrayCollection(usersGrid.dataProvider))"/>
    <mx:Button label="Clear"
        click="usersOfD.clearCache(ArrayCollection(usersGrid.dataProvider))" />
    <mx:Script>
        <![CDATA[
            [Bindable]
            private var users:ArrayCollection;
            ...
            private function loadData(){
                usersOfD.fill(users);
            }
        ...     ]]>
    </mx:Script>
</mx:Application>
```

In the preceding example, the `users` are loaded when the application is initialized by calling the `fill()` method of the `usersOfD` service. Clicking the Save button calls the `saveCache()` method to pass in the `users` ArrayCollection. The items of the collection are stored to the local store.

Clicking the Clear button calls the `clearCache()` method to pass in the same `users` ArrayCollection. The items of the collection are removed from the local store. Since this collection was the only data stored locally, no data remains in the local store.

Disabling connection attempts and performing local fills

You can use the combination of the `DataService.autoConnect` property and the `DataService.fallBackToLocalFill` property to disable connection attempts and perform local fill requests.

Set the `autoConnect` property to `false` so that operations that require a connection do not fault when the `DataService` instance is disconnected unless there is problem loading data from the local cache. When you also set the `fallBackToLocalFill` property to `true`, the `DataService` instance makes local fill requests when not connected to the server.

The following `creationComplete()` event handler for a DataGrid control sets the `autoConnect` property to `false` and the `fallBackToLocalFill` property to `true`.

Note: For an AIR-based client that uses a custom offline adapter, you would set the `DataService.offlineAdapter` property in addition to the properties shown here. For more information, see “[Using an offline adapter with an AIR-based client](#)” on page 317.

```
< mx:Script>
    <! [CDATA [
        ...
        protected function
            dataGrid creationCompleteHandler(event:FlexEvent) :void
        {
            // Create a DataService instance.
            var myDataService:DataService = new DataService("destination1");
            // Set the unique ID for the local data store.
            myDataService.cacheID = "cache1";
            // Save the cache automatically when online.
            myDataService.autoSaveCache = true;
            // If working offline don't attempt to connect to the server.
            myDataService.autoConnect = false;
            // Query against the local store when not connected.
            myDataService.fallBackToLocalFill=true;
            // Perform a fill operation.
            myDataService.fill(users, "notorious");
        }
    ]]>
</mx:Script>
```

Request parameters identify the items in a request. In the preceding example, the key for the list of items returned in the `fill` request is `notorious`. When the application is reloaded in a disconnected state, it requests the items with the key value of `"notorious"` from the local store.

The `DataService.connect()` method corresponds to the `autoConnect` property. The `connect()` method lets you control the timing of a reconnect attempt when the `autoConnect` property is set to `false`. The `DataService` instance always attempts to load any local data before attempting to establish a connection and satisfy a `fill()` or `getItem()` request. Setting the `autoConnect` property to `false` prevents any attempts to connect with the remote destination. Changing the value to `true` is equivalent to calling the `connect()` method. This behavior lets you use data binding to establish the connectivity of an application using a user interface component such as a CheckBox.

If there is a connection available after the local cache is loaded, the current value of the `reconnectPolicy` property determines how a request for the current data is made. If the `reconnectPolicy` property is set to `IDENTITY`, no request for remote data is made because it is assumed that the data is up-to-date. If the `reconnectPolicy` property is set to `INSTANCE`, a request for the remote data is made and the result of that fill is used to overwrite the current in-memory version. In this situation, if the `autoSaveCache` property is set to `true` when the new data is returned, it is saved to local storage overwriting the previous version.

For paged and lazily loaded data, each time a request is satisfied, the results are stored locally when the `autoSaveCache` property is set to `true`. To make all of the data available offline for a paged result set, call the `createCursor()` method of the `ArrayCollection` you using and then call the cursor's `seek()` method, specifying a `prefetch` value of the entire length.

When a client tries to commit stale data from changes made offline, the assembler on the server can use the old and new versions of the object to report a conflict.

Note: *Run-time configuration information is saved to the local cache when present during a save, and is restored during initialization if a connection cannot be established. For more information about run-time configuration, see “[Run-time configuration](#)” on page 411.*

Inspecting the data cache

You use the `mx.data.CacheDataDescriptor` class to inspect the various attributes of offline cached data. This class contains the following properties: `lastAccessed:Time`, `lastWrite:Time`, `creation:Time`, `metadata:Object`, and `id:Object`.

The `DataService` class contains the following related methods: `getCacheDescriptors()`, `getCacheData()`, and `clearCacheData()`.

The `getCacheDescriptors()` method returns an `ICollectionView` of `CacheDataDescriptor` instances describing the attributes for all of the cached data. If a managed `ArrayCollection` or a managed item is specified, the collection returned contains only a single `CacheDataDescriptor` specific to the specified argument. The `getCacheData()` method returns the collection or item data based on the specified cache descriptor. The returned value is pulled directly from the local cache and is not managed. Additionally, a new instance of the collection or object is returned on each call. Calling the `clearCacheData()` method removes the associated data from the local store for the descriptor specified.

The following example shows an application that inspects the data cache. In this example, all of the cached collection and item information is returned for the `usersOFD` service. The `metadata` property is set as a string that contains a descriptive name for the cached data. The grid displays the descriptive name along with the last accessed, creation, and last updated times.

```
<mx:Application creationComplete="displayCacheInfo() ">
    <mx:DataService id="usersOfD" destination="D" />
    <mx:DataGrid id="cachedDataInfo" dataProvider="{cacheInfo}"
        change="updateDisplay() ">
        <mx:columns>
            <mx:DataGridColumn headerText="Name" dataField="metadata" />
            <mx:DataGridColumn headerText="Last Accessed"
                dataField="lastAccessed"/>
            <mx:DataGridColumn headerText="Created" dataField="created" />
            <mx:DataGridColumn headerText="Last Updated" dataField="lastWrite"/>
        </mx:columns>
    </mx:DataGrid>
    <mx:DataGrid id="dataDisplay" dataProvider="{cacheData}"/>
    <mx:ArrayCollection id="cacheData"/>
    <mx:Script>
        <![CDATA[
            [Bindable]
            private var cacheInfo:ArrayCollection;
            private function displayCacheInfo(){
                usersOfD.getCacheDescriptors(cacheInfo, CacheDescriptor.FILL);
            }
            private function updateDisplay():void{
                var token:AsyncToken =
                    usersOfD.getCacheData(cachedData.selectedItem);
                token.addResponder(new AsyncResponder(
                    function (event:ResultEvent, o:Object):void{
                        cacheData = ArrayCollection(event.token.result);
                    },
                    function (event:FaultEvent, o:Object):void{
                        Alert.show("Could not load cache data");
                    }));
            }
        ]]
    </mx:Script>
</mx:Application>
```

The following example shows an application in which all of the cached collection and item information is returned for the `usersOfD` service. The `metadata` property is set to a string that contains a descriptive name for the cached data. The DataGrid displays the descriptive name along with the last accessed, creation, and last updated times. When the user selects an item and clicks the Remove button, the selected item (`CacheDescriptor`) is used first to access the existing data from the cache if it is not already loaded, and then to pass that item to the `clearCacheData()` method, which removes it from the local cache.

```
<mx:Application creationComplete="displayCacheInfo() ">
    <mx:DataService id="usersOfD" destination="D" />
    <mx:DataGrid id="cachedData" dataProvider="{cacheInfo}">
        <mx:columns>
            <mx:DataGridColumn headerText="Name" dataField="metadata" />
            <mx:DataGridColumn headerText="Last Accessed"
                dataField="lastAccessed"/>
            <mx:DataGridColumn headerText="Created" dataField="created"/>
            <mx:DataGridColumn headerText="Last Updated" dataField="lastWrite"/>
        </mx:columns>
    </mx:DataGrid>
    <mx:Button label="Remove"
        click="removeFromCache(cachedData.selectedItem) "/>
    <mx:Script>
        <![CDATA[
            [Bindable]
            private var cacheInfo:ArrayCollection;
            private function displayCacheInfo(){
                usersOfD.getCacheDescriptors(cacheInfo);
            }
            private function
                removeFromCache(descriptor:CacheDataDescriptor):void{
                    var token:AsyncToken = usersOfD.getCacheData(descriptor);
                    token.responder = new AsyncResponder(
                        function (result:ResultEvent, o:Object):void{
                            usersOfD.clearCacheData(descriptor);
                        },
                        function (fault:FaultEvent, o:Object):void{
                            Alert.show("Failed to get specified cache data");
                        }));
            }
        ]]>
    </mx:Script>
</mx:Application>
```

Using an offline adapter with an AIR-based client

To get the best occasionally connected client experience, use an AIR-based client with a custom SQLite offline adapter. In the custom offline adapter, you write logic for SQL queries that retrieve locally cached items in a way similar to how a Data Management Service assembler retrieves items from the data source on the server.

When using a SQLite offline adapter, high-level offline persistence behavior, such as how fills are replayed, is the same as in LiveCycle Data Services 3. However, the persistence format differs for AIR clients. Instead of persisting each item as a BLOB on each row in the item's SQLite table, each has a column for each of its properties. This makes it easier to view and manipulate client-side data.

To migrate to the SQLite adapter functionality, existing clients must recreate their caches. For migrated clients, it is possible that differences in persistence can lead to subtle differences in behavior. If this happens, you can revert to the Livecycle Data Services 3 persistence format by making a declaration on the DataService instance to explicitly set the offline adapter to a DataServiceOfflineAdapter instance instead of a SQLiteOfflineAdapter instance, as the following code shows:

```
DataService.offlineAdapter = new DataServiceOfflineAdapter();
```

Note: You can use model-driven development to build this type of client; for more information, see “[Building an offline-enabled application](#)” on page 333.

The custom offline adapter must extend the mx.data.SQLiteOfflineAdapter class, which provides SQLite offline query capabilities for AIR clients that use data management. The custom class must override the `getQueryCriteria()` method and can also override the `getQueryCriteriaParameters()` and `getQueryOrder()` methods of the SQLiteOfflineAdapter class.

The following table describes the SQLiteOfflineAdapter methods.

Method	Description
<code>getQueryCriteria()</code>	Defines the SQL WHERE clause.
<code>getQueryCriteriaParameters()</code>	Defines the parameters of SQL WHERE clause.
<code>getQueryOrder()</code>	Defines the SQL ORDERBY clause.

The SQLite table for a particular destination is named `Entity_destination`. The table column names match the entity property names, unless they conflict with a SQLite keyword, in which case a `_` is prepended to the name.

The following example shows a custom offline adapter named `ProductOfflineAdapter` that overrides all three SQLiteOfflineAdapter methods:

```
package com.adobe.offline
{
import mx.data.SQLiteOfflineAdapter;
import mx.utils.StringUtil;
import mx.core.mx_internal;
public class ProductOfflineAdapter extends SQLiteOfflineAdapter
{
    override protected function getQueryCriteria(originalArgs:Array):String
    {
        var queryCriteria:String;

        var args:Array = originalArgs.concat();
        var filterName:String = args.shift();
        var accessedFieldNames:Array;
        switch (filterName)
        {
            case "getAll":
                break;
            case "getByProductid":
                return "productid = :productid";
                break;
            case "getByDescription":
                return "description = :description";
                break;
            case "getByPrice":
                return "price = :price";
                break;
            case "getByProductName":
                return "name = :productname";
                break;
        }
        if (!queryCriteria)
        {
            queryCriteria = super.getQueryCriteria(originalArgs);
        }
        return queryCriteria;
    }
}
```

```
override protected function getQueryCriteriaParameters(originalArgs:Array):Object
{
    var queryCriteriaParameters:Object;
    var args:Array = originalArgs.concat();
    var filterName:String = args.shift();
    switch (filterName)
    {
        case "getAll":
            break;
        case "getByProductid":
            queryCriteriaParameters = {":productid"};
            break;
        case "getByDescription":
            queryCriteriaParameters = {":description"};
            break;
        case "getByPrice":
            queryCriteriaParameters = {":price"};
            break;
        case "getByProductname":
            queryCriteriaParameters = {":productname"};
            break;
    }
    if (!queryCriteriaParameters)
    {
        queryCriteriaParameters = super.getQueryCriteriaParameters(originalArgs);
    }
    return queryCriteriaParameters;
}
override protected function getQueryOrder(originalArgs:Array):String
{
    var queryOrder:String;

    var args:Array = originalArgs.concat();
    var filterName:String = args.shift();
    var accessedFieldNames:Array;
    switch (filterName)
    {
        case "getAll":
            break;
        case "getByProductid":
            break;
        case "getByDescription":
            break;
        case "getByPrice":
            break;
        case "getByProductname":
            break;
    }
    if (!queryOrder)
    {
        queryOrder = super.getQueryOrder(originalArgs);
    }
    return queryOrder;
}
```

The following example shows an AIR-based client that uses a custom offline adapter. For information on building a complete model-driven application that uses this client MXML file, see “[Building an offline-enabled application](#)” on page 333.

In the MXML file, the lines in bold highlight code that is specific to this type of application.

- Import the offline adapter. In this case, the adapter is in the com.adobe.offline package.
- Create a variable for the offline adapter. In this case, the variable is `myOfflineAdapter`.
- Assign the `myOfflineAdapter` variable to the `offlineAdapter` property of the `DataService` instance, which in this case is `productService.serviceControl`.

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
xmlns:mx="library://ns.adobe.com/flex/mx" xmlns:OfflineAIRAPP="OfflineAIRAPP.*"
preinitialize="app_preinitializeHandler(event)"
creationComplete="windowedapplication1_creationCompleteHandler(event)">
    <fx:Script>
        <![CDATA[
            import com.adobe.offline.ProductOfflineAdapter;

            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.messaging.Channel;
            import mx.messaging.ChannelSet;
            import mx.messaging.channels.RTMPChannel;
            import mx.messaging.events.ChannelEvent;
            import mx.rpc.AsyncToken;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            public var myOfflineAdapter:ProductOfflineAdapter;
            public function channelConnectHandler(event:ChannelEvent):void
            {
                productService.serviceControl.autoConnect=false;

            }
            protected function
                app_preinitializeHandler(event:FlexEvent):void
            {
                var cs:ChannelSet = new ChannelSet();
                var customChannel:Channel = new RTMPChannel("my-rtmp",
                    "rtmp://localhost:2037");
                cs.addChannel(customChannel);
                productService.serviceControl.channelSet=cs;
                customChannel.addEventListener(ChannelEvent.CONNECT,
                    channelConnectHandler);
            }

            protected function dataGrid_creationCompleteHandler(event:FlexEvent):void
            {
                getAllResult.token = productService.getAll();
            }

            protected function
                windowedapplication1_creationCompleteHandler(event:FlexEvent):void
```

```
{  
    productService.serviceControl.autoCommit = false;  
    productService.serviceControl.autoConnect = false;  
    productService.serviceControl.autoSaveCache = false;  
    productService.serviceControl.offlineAdapter = new  
        ProductOfflineAdapter();  
    productService.serviceControl.fallBackToLocalFill = true;  
    productService.serviceControl.encryptLocalCache = true;  
    productService.serviceControl.cacheID = "myOfflineCache";  
}  
  
protected function connectBtn_clickHandler(event:MouseEvent):void  
{  
    productService.serviceControl.connect();  
}  
  
protected function DisconnectBtn_clickHandler(event:MouseEvent):void  
{  
    productService.serviceControl.disconnect();  
}  
  
protected function commitBtn_clickHandler(event:MouseEvent):void  
{  
    productService.serviceControl.commit();  
}  
  
protected function saveCacheBtn_clickHandler(event:MouseEvent):void  
{  
    productService.serviceControl.saveCache();  
}  
  
]]>  
</fx:Script>  
<fx:Declarations>  
    <s:CallResponder id="getAllResult" />  
    <OfflineAIRAPP:ProductService id="productService"  
        fault="Alert.show(event.fault.faultString + '\n' +  
        event.fault.faultDetail)"/>  
</fx:Declarations>  
<mx:DataGrid editable="true" x="9" y="10" id="dataGrid"  
    creationComplete="dataGrid_creationCompleteHandler(event)"  
    dataProvider="{getAllResult.lastResult}">
```

```
<mx:columns>
    <mx:DataGridColumn headerText="productid" dataField="productid"/>
    <mx:DataGridColumn headerText="description" dataField="description"/>
    <mx:DataGridColumn headerText="price" dataField="price"/>
    <mx:DataGridColumn headerText="productname" dataField="productname"/>
</mx:columns>
</mx:DataGrid>
<s:Button x="10" y="246" label="Connect" click="connectBtn_clickHandler(event)" id="connectBtn" width="84" height="30"/>
<s:Button x="112" y="204" label="Save to Local Cache" id="saveCacheBtn" click="saveCacheBtn_clickHandler(event)" height="30"/>
<s:Button x="110" y="246" label="Commit to Server" id="commitBtn" click="commitBtn_clickHandler(event)" width="135" height="30"/>
<s:Button x="10" y="204" label="Disconnect" id="DisconnectBtn" click="DisconnectBtn_clickHandler(event)" height="30"/>
<s:Label x="270" y="204" text="{'Commit Required: ' + productService.serviceControl.commitRequired}"/>
<s:Label x="270" y="246" text="{'Connected: ' + productService.serviceControl.connected}"/>
</s:WindowedApplication>
```

The `windowedapplication1_creationCompleteHandler()` method sets the following properties that affect offline behavior:

Property	Description
<code>autoCommit</code>	Determines if the application automatically commits data changes.
<code>autoConnect</code>	Determines if the application automatically connects to the server when a connection is available.
<code>autoSaveCache</code>	Determines if data is automatically saved to the offline cache.
<code>offlineAdapter</code>	Assigns a custom offline adapter.
<code>fallBackToLocalFill</code>	Determines if the application falls back to using local fills from the AIR SQLite database when not connected to the server.
<code>encryptLocalCache</code>	Set this property to true to encrypt the SQLite database and ensure that data is unavailable to all other users and applications running on the same machine. The database is encrypted with a random encryption key and password using the SQLite encryption feature. The password is stored locally using AIR EncryptedLocalStore support, associated with the database name.
<code>cacheID</code>	Assigns a cache ID for storing the local copy of data in the AIR SQLite database.

About caching with Flash Player and AIR

Flash Player and AIR each cache data to the local disk. Flash Player uses local shared objects (LSOs), while AIR uses a SQLite database.

Flash Player creates a .sol file to store the LSO. Flash Player creates a new directory for the application and domain based on your platform and operating system. For example, for a Flash Player application on Microsoft Windows, the .sol files are created in the under the following directory:

```
c:/Documents and Settings/userName/Application Data/Macromedia/Flash Player/#SharedObjects
```

Note: There are performance and scalability limitations when using LSOs to cache data offline. There is a configurable user limit, with a default size of 100 kilobytes. Writes are monolithic, so each save writes all managed data for that data store.

When using AIR, SQL database files are created in the `_ds_localcache` subdirectory of the application storage directory. The location of the application storage directory is defined by the setting of the `air.File.applicationStorage` property. For example, for an AIR application named TestApp running on Windows, SQL database files are created in the following directory:

```
C:/Documents and Settings/userName/Application Data/TestApp/Local Store/_ds_localstore
```

When using AIR, each `DataService.cacheID` value corresponds to a single database file.

When you compile your application, you must link in the appropriate SWC file to control caching, depending on whether you are deploying your application on Flash Player or AIR, as follows:

- `playerfds.swc` Link this SWC file for applications running in Flash Player.
- `airfds.swc` Link this SWC file for applications running in AIR.

Server push

Use the server-side `DataServiceTransaction` class to push changes to managed data stored on clients.

Using server push APIs

Use the `flex.data.DataServiceTransaction` class to push server-side data changes made outside the Data Management Service to Data Management Service clients when your code is running within the same web application as LiveCycle Data Services. The changes are injected into the Data Management Service as if they were initiated in the system. The `DataServiceTransaction` class is documented in the LiveCycle Data Services Javadoc API documentation.

Use a `DataServiceTransaction` instance from server-side code outside the Data Management Service to push changes to managed data stored on clients that have their client-side `DataService` object's `autoSyncEnabled` property set to `true` or have subscribed with matching criteria using the `manualSync` property.

There are two distinct use cases for the `DataServiceTransaction` class:

- Use the instance of `DataServiceTransaction` that is created for each assembler operation that modifies the state of objects that the Data Management Service manages.
- Create an instance of `DataServiceTransaction` in your own code.

For the first use case, (use the `DataServiceTransaction` that is created when an Assembler operation is called), call its static `getCurrentDataServiceTransaction()` method. Next call its `updateItem()`, `deleteItem()`, or `createItem()` method to trigger additional changes. Call these methods to apply changes you have persisted or will be persisting in this transaction. If the current transaction is rolled back, these changes are not pushed to clients.

As an example of how you can push data when you already have an instance of DataServiceTransaction, consider a JSP page that updates a database and a Data Management Service destination that uses the same database. Flex clients connected to the Data Management Service destination normally are not notified of changes that the JSP page makes to the database. However, you can modify the JSP page to call a data access object (DAO) in the data-managed application when it makes database changes. The DAO in turn calls the `createItem()` method on the assembler, which automatically creates a DataServiceTransaction instance. Immediately following the `createItem()` method success, you could include code to call the `getCurrentDataServiceTransaction()` of the DataServiceTransaction instance and then call its `createItem()` method.

For the second use case (you do not already have a DataServiceTransaction instance), you can call the static `DataServiceTransaction.begin()` method to initiate a transaction, as the following example shows:

```
...
public void createBook() {
    DataServiceTransaction dtx = DataServiceTransaction.begin(false);
    // Create a product:
    Product prod = new Product();
    prod.setProductName("Product1");
    prod.setDescription("A great product");
    prod.setPrice(10);
    ProductDAO dao = new ProductDAO();
    dao.create(product);
    //Inform the Data Management Service:
    dtx.createItem("product", prod);
    dtx.commit();
}
```

To roll back a transaction when working with a DataServiceTransaction instance, you mark the `javax.transaction.UserTransaction` instance as rolled back as you would in a normal J2EE application, or you can call the `setRollbackOnly()` method on the DataServiceTransaction.

The DataServiceTransaction class provides access to the current transaction in a thread local state, and holds messages to be pushed to clients when the transaction completes. You also use this class to register for synchronization events before and after completion of the transaction.

Each DataServiceTransaction instance is stored in thread-local state and it is assumed that it is only operated on one thread at a time. It is not thread safe.

If you use the DataServiceTransaction class from within a sync, update, create, or delete method of your assembler, do not use this class to indicate changes made to items that are already in the midst of being changed in this transaction. Doing so queues an additional change to that object instead of modifying the currently active one, which can create a conflict. Instead, you update the NewVersion instance with your changed property values and add any newly changed property values to the list of changed properties sent to you. For example, if after every update made to an instance, you want to change the `versionId` of that instance, you add the `versionId` to the list of changes and also update the `versionId` value in your newVersion instance.

If you are using the ChangeObject interface, you call the `addChangedPropertyName()` method to add the `versionId` property. If you are using the `updateItem()` method, you just add that property to the list provided to your `updateItem()` method.

Refreshing fills from server code

The `DataServiceTransaction.refreshFill()` method lets you manually refresh a fill or matching set of fills from server code either as part of your assembler's sync method or from other server-side code. You specify a list of fill parameters that are used to create a matching list of fills that are currently being cached by active clients. This list can be null, which means that all fills on that destination match. If the list is non-null, the `refreshFill()` matches fills made by clients with the same number of parameters if all of the slots match based on the rules in the following table:

Value	Rule
Null value	Matches that slot unconditionally.
Class value	Matches a parameter in the slot of that type.
Any other value	Matches fill parameters by using the equals method.

There are two variants of the `refreshFill()` method. The simpler variant takes a destination and fill parameters as its two parameters. The other variant takes an additional parameter that specifies a `PropertySpecifier` instance. This variant compares the properties of all of the items in the collection. It could use the item cache to avoid updating all items if the item cache is enabled. It compares the properties of all items in the new version of the fill and sends update messages for any properties that have changed. If the item cache is disabled, this `refreshFill()` method sends update item events for all items in the collection just in case they have changed.

The `DataServiceTransaction` class also has `addItemToFill()` and `removeItemFromFill()` methods that you can use to make an explicit change to a filled collection actively managed by other clients. Use of these methods generates an update collection message in the `DataServiceTransaction` that is sent to the committing client, any clients that have called `fill` on that collection, and any clients that have subscribed to this change using the manual sync mode properties set in the `DataServiceTransaction`. This is an efficient way to modify a managed collection on clients without re-executing the entire query.

Note: When you compile code that uses LiveCycle Data Services Java APIs, you must include the `flex-messaging-data.jar` and `flex-messaging-common.jar` files in your classpath.

Chapter 7: Model-driven applications

Adobe application modeling technology is a set of technologies that facilitates the development of data-centric applications. Use application modeling technology with Flash Builder to generate code based on a data model for communicating with Data Management Service and RPC service destinations. Generated client code for Data Management Service destinations is built on the DataService object. Generated client code for RPC services is built on ActionScript objects such as RemoteObject and WebService objects.

Two types of model-driven development

There are two types of model-driven development with LiveCycle Data Services:

- The first type is end-to-end model-driven development for both client and server code. You control the application code on the client and server by customizing the model from which code is generated.
- The second type is model-driven development of client code you can use with existing server destinations on the LiveCycle Data Services server. In this case, you generate a model and the corresponding client-side ActionScript code by introspecting a destination on the LiveCycle Data Services server. This type of development is useful when you have existing destinations. However, it gives you no control over the server implementation and very little control over the client implementation. You can make only limited changes to the model, and those changes affect only client code generation.

When you perform end-to-end model-driven development, you can generate Data Management Service destinations based on a model without writing any Java code. The generated server code is based on the Assembler interface of the Data Management Service and uses the Hibernate object/relational persistence and query service. The generated server code works in combination with the generated client code in Flash Builder.

Use the following software to build an end-to-end model-driven applications with LiveCycle Data Services:

- LiveCycle Data Service server
- Flash Builder
- Application modeling plug-in for Eclipse (the Modeler)
- SQL database configured as a JDBC data source (Model Assembler feature only)

For general information about application modeling technology, see the [Application Modeling Technology Reference](#).

The role of Flash Builder in model-driven development

Flash Builder uses application modeling technology in features that simplify the development of data-centric Flex client applications. Flash Builder generates client code for calling remote services.

The data-centric development features in Flash Builder depend on an application modeling technology model from which Flash Builder generates the ActionScript code for calling a remote service.

Building your first model-driven application

This set of tasks builds an end-to-end model-driven application that uses the lcds-samples web application and the ordersdb SQL database on the LiveCycle Data Services server. These tasks are specific to the LiveCycle Data Services installation with an integrated Apache Tomcat application server. Application-server-specific configuration is noted where applicable.

End-to-end model-driven development lets you control the client and server code for your application by customizing the model from which code is generated. With just Flash Builder, the Modeler, and the LiveCycle Data Services server, you can quickly build an end-to-end data-centric application.

Summary of steps

Perform the following tasks to build a basic end-to-end model-driven application:

- 1 Install [LiveCycle Data Services](#), [Flash Builder](#), and the [application modeling plug-in](#).
- 2 Configure a JDBC data source.
- 3 Configure Remote Data Services (RDS) on the server.
- 4 Configure Flash Builder to use RDS.
- 5 Configure server-side debug logging.
- 6 Create a J2EE server project in Flash Builder.
- 7 Build a model and generate code.
- 8 Create and run a Flex client.

Configure a JDBC data source

An end-to-end model-driven application depends on a JDBC data source configured on the application server. In the data source configuration, you reference the relational database for which you want to build a model-driven application.

On the Tomcat server, you configure JDBC data sources in context files. On the LiveCycle Data Services installation with an integrated Tomcat server, the data source for this application is preconfigured in the lcds-samples.xml context file located in the following directory:

install_root/tomcat/conf/Catalina/localhost

where *install_root* is the directory where you installed LiveCycle Data Services. Within the lcds-samples.xml file, the following data source is preconfigured for the lcds-samples web application. How you configure a JDBC data source depends on the application server you use. For more information, see your application server documentation.

```
<Context privileged="true" antiResourceLocking="false">
    antiJARLocking="false" reloadable="true">
        <Resource name="jdbc/ordersDB" type="javax.sql.DataSource"
            driverClassName="org.hsqldb.jdbcDriver"
            maxIdle="2" maxWait="5000"
            url="jdbc:hsqldb:hsq1://localhost:9002/ordersdb"
            username="sa" password="" maxActive="4"/>
</Context>
```

Note: The database driver classes required to connect to the sample database are pre-installed in the *install_root/tomcat/lib/hsqldb.jar* file for the LiveCycle Data Services installation with integrated Tomcat server. If you are using a different configuration, ensure that you have the database driver classes in your classpath.

More Help topics

[“Configuring a data source” on page 371](#)

Configure RDS on the server

To successfully retrieve data from the data source, enable the RDS server (the RDSDispatchServlet servlet) in the lcds-samples web application. To perform this task, edit the web.xml file in an XML editor or text editor. The web.xml file for the lcds-samples web application is located in the following directory:

install_root/tomcat/webapps/lcds-samples/WEB-INF/web.xml

where *install_root* is the directory where you installed LiveCycle Data Services.

In the web.xml file, remove the comments from the RDSDispatchServlet definition. Set the `useAppserverSecurity` parameter value to `false`.

```
<servlet>
    <servlet-name>RDSDispatchServlet</servlet-name>
    <display-name>RDSDispatchServlet</display-name>
    <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-class>
        <init-param>
            <param-name>useAppserverSecurity</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>10</load-on-startup>
    </servlet>
    <servlet-mapping id="RDS_DISPATCH_MAPPING">
        <servlet-name>RDSDispatchServlet</servlet-name>
        <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
    </servlet-mapping>
```

Note: For local development, you can use the RDSDispatchServlet servlet without configuring application server security, as shown above. Configure application server security for all other deployments.

More Help topics

[“Configuring RDS on the server” on page 371](#)

Configure server-side debug logging

During development, it is useful to set server-side logging to the debug level and capture messages related to model-driven applications. While you are running model-driven Flex clients, you can view the related server traffic in the application server console window.

To see log messages related to model-driven applications, replace the logging section of the lcds-samples/WEB-INF/flex/services-config.xml file with the following section:

```
<logging>
    <!-- You may also use flex.messaging.log.ServletLogTarget -->
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[LCDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>true</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <!--<pattern>Endpoint.*</pattern>-->
            <!--<pattern>Service.*</pattern>-->
            <pattern>Service.Data.Fiber</pattern>

            <pattern>Message.*</pattern>
            <pattern>DataService.*</pattern>
            <pattern>Configuration</pattern>
        </filters>
    </target>
</logging>
```

Configure Flash Builder to use RDS

After you configure RDS on the server, configure Flash Builder to connect to the RDS server.

Complete the following steps to configure Flash Builder for RDS:

- 1 Start your application server if it isn't already running.
 - 2 In Flash Builder, select Window > Preferences.
 - 3 Expand the Adobe tree node and select RDS Configuration.
 - 4 Under Currently Configured RDS Servers, select LCDS (localhost).
 - 5 Enter a description and the hostname and port number of your RDS server. For this application, use the default hostname value (127.0.0.1) and the default port value (8400).
 - 6 Enter the context root of the web application that hosts your RDS server. For this application, use the default value (lcds-samples).
 - 7 Leave the password field blank and select Prompt for Password.
- Note: You are not using web application security for this application. If you were using web application security, you would enter a valid username and password combination.*
- 8 Click Test Connection.
 - 9 If the connection is successful, click OK.

Create a Flex project in Flash Builder

To use Flash Builder with LiveCycle Data Services, create a Flex project that corresponds to the server and web application you are using. This project references the J2EE application server hosting LiveCycle Data Services. That is, when you create the project, select J2EE as the Application Server type. Ensure that the project references the context root where you created the data source. After you create the project, all of the client libraries required to interact with the J2EE application server are automatically added to your project's class path.

Complete the following steps to create a Flex project:

- 1 Start your application server if it isn't already running.
- 2 Start Flash Builder if it isn't already running.
- 3 In Flash Builder, select File > New > Flex Project.
- 4 Enter a project name.
- 5 Use the default project location.
- 6 Set the application type as Web.
- 7 Select the Flex SDK version you want to use. You can use the default Flex SDK or Flex SDK 3.5.
- 8 In the Application Server list, select J2EE.
- 9 Make sure Use Remote Object Access Service is checked.
- 10 Make sure the LiveCycle Data Services checkbox is checked.
- 11 Click Next.
- 12 Enter the server location information for your server.
- 13 Click Validate Configuration.
- 14 If the configuration is valid, click Finish to create the project.

Build a model and generate code

- 1 Start the samples database by running the startdb.bat or startdb.sh file, depending on your operating system. These files are located in the following directory:

install_root/sample_db

where *install_root* is the directory where you installed LiveCycle Data Services.

- 2 Start your application server if it isn't already running.
- 3 Start Flash Builder if it isn't already running.
- 4 In Flash Builder, make sure that you have created a Flex J2EE server project and it is the active project.

Note: To make a project active, open an application file in the project and make it the active file in the editor.

- 5 In the Package Explorer, click the Filters button. In the Filters dialog, deselect ".model". This configuration lets you view model files in the Flash Builder Package Explorer.
- 6 In the Package Explorer, click the Open Model for Active Project button located at right end of the toolbar. Clicking this button for the first time creates a directory named .model and a model file with the same name as the Flex project. The model file opens automatically.

After the model is created, clicking the Open Model for Active Project button opens the model file for editing. The same button also appears in the toolbar for the Data/Services view when in the Flash perspective.

- 7 Switch to the Data Model perspective (the Modeler): click the Open Perspective button in the right corner of the Flash Builder window and then select Data Model.
- 8 Select the Design tab to enter the Design view.
- 9 By default, the RDS Data view is in the lower right side of the Data Model perspective. In the RDS Data view, expand the RDS server tree nodes to view the tables in the java:/comp/env/jdbc/ordersDB data source.

10 Select all of the tables in the ordersDB data source and drag them to the Design view canvas. An entity is added to the model for each database table. The entities are represented in a diagram that is similar to a UML (Unified Modeling Language) diagram.

11 Save the model and click the Deploy Model to LCDS server button. Accept the default settings and click OK.

Clicking OK generates a Model Assembler destination on the server for each entity. You can call these destinations from a Flex client to work with data stored in the database. By default, the model file is saved in the WEB-INF/datamodel directory of the web application to which you deploy.

In the application server console, you can see a ProductAssembler class is instantiated after you deploy the model.

12 Switch back to the Flash perspective. You should see a new service in the Data/Services view. Based on the model you created, Flash Builder generates ActionScript classes that provide access to service operations from a client application. The operations for the service are available in the Data/Services view.

Updating database tables when you deploy models

When you redeploy a model to the server with the Update radio button selected, verify that your database tables are updated correctly. There are situations where the Hibernate does not drop table columns even though you have removed the corresponding properties from entities in the model. This happens when a database owner (dbo) name is a prefix in the table name; for example, PUBLIC.EMPLOYEE, where PUBLIC is the dbo name. There is no indication from Hibernate or the Modeler that the update failed. You can work around this issue by removing the dbo name from the table name. When looking for a table, Hibernate prepends the dbo (PUBLIC, in this case) to the table name. If the table name in the model is already PUBLIC.EMPLOYEE, Hibernate looks for PUBLIC.PUBLIC.EMPLOYEE and fails to find it.

Another fact to be aware of is that the Modeler cannot perform tasks that your database management system cannot perform. Issues can arise when you redeploy a model to the server with the Update radio button selected when you changed the data type of an entity property. The changes may not be reflected properly in the database tables on the server if the entity you changed is used as a property in another entity.

For example, suppose you have an entity called State with stateId integer and stateName string properties. You also have a Customer entity with a state property with a data type of State. You first deploy the model to create the State database table and update the Customer table to reference the State table as a foreign key. You then change the stateId property of the State entity from integer to string. When you redeploy the model with the Update radio button selected, the State table still uses integer for stateId even though you changed the model to make stateId a string.

Create and run a Flex client

To use the client and server code generated from the model, create a Flex web client that has a simple master-detail user interface.

Complete the following steps to create a Flex client:

- 1** Start the samples database if it isn't already running.
- 2** Start your application server if it isn't already running.
- 3** Start Flash Builder if it isn't already running.
- 4** In Flash Builder, switch to Design view in the Flash perspective.
- 5** In the Package Explorer, expand the nodes of your Flex project to src > (default package) and open the displayed MXML file.
- 6** Select the Components tab in the lower left side of the Flash Builder window.
- 7** Expand the Controls folder in the Components tree.

- 8 Drag the DataGrid control to the Design view canvas.
- 9 Select the Data/Services tab under the Design view canvas.
- 10 In the Data/Services view, expand the ProductService node.
- 11 Drag the `getAll() : Product []` function to the DataGrid control. Accept the default Bind To Data settings. The DataGrid columns changes to match the property names of the Product entity.
- 12 Right-click the DataGrid control and select Generate Details Form.
- 13 Select Model Driven Form and click OK to generate a Form.
- 14 Reposition the form so that it is not covering the DataGrid control.
- 15 Run the application. The DataGrid control should fill with data; the `productService.getAll()` function is called in the DataGrid control's creationComplete event handler.

Note: Calling the `getAll()` function results in a call to the `fill()` method of the underlying DataService instance.
- 16 Select a row in the DataGrid control to display details in the form.
- 17 Click Delete to delete an item from the database.

Note: Deleting an item with the form results in a call to the `deleteItem()` method of the underlying DataService instance.
- 18 Click Add, enter data for a new item in the form, and then click Save to add an item to the database.

Note: Adding an item with the form results in a call to the `createItem()` method of the underlying DataService instance.

Working with dynamic destinations in model-driven applications

Model-driven applications use dynamic destinations; for more information, see “[Accessing dynamic components with a Flex client application](#)” on page 417. As demonstrated in this application, you do not always have to explicitly assign a channel in the Flex client or in the model. If you do not specify a channel set and channel when you use a dynamic destination, LiveCycle Data Services attempts to use the default application-level channel assigned in the `defaultChannels` element in the `services-config.xml` file.

If you rely on application-level default channels, you must make sure that those channels are the ones expected by the destination you are using. For example, if the dynamic destination is created without specifying any channels on the server, it implicitly expects connections only over the service-level default channels. This is fine if service-level and application-level channels match, but if service-level default channels are different than application-level default channels, the client cannot contact the destination using the application-level default channels. In that case, a channel set with the proper channel is required on the client and you cannot rely on the application-level default channels.

Accessing DataService properties and methods in model-driven applications

In model-driven Flex clients that use the Data Management Service, the client-side `mx.data.DataService` component is inside a generated service wrapper instance. The service wrapper is an instance of the `com.adobe.fiber.services.wrapper.DataServiceWrapper` class. This is different than a typical Flex client that calls a Data Management Service, for which you usually instantiate a `DataService` directly in an MXML element or in the script block of an MXML file.

In model-driven development, the `DataService` instance is accessed as the `serviceControl` property of the service wrapper. You have the same access to the `DataService` members that you have for non-model-driven clients. You set `DataService` properties and methods by using the following dot notation in your ActionScript code:

- `xxxService.serviceControl.propertyName`
- `xxxService.serviceControl.methodName()`

So, for example, to save data to a local cache (offline), set the `DataService.cacheID` property as follows:

```
protected function dataGrid_creationCompleteHandler(event:FlexEvent):void
{
...
    productService.serviceControl.cacheID="cache2";
...
}
```

More Help topics

[“Building the client application” on page 372](#)

Building an offline-enabled application

You can use an offline adapter with an AIR SQLite database to perform offline fills when a desktop client is disconnected from the LiveCycle Data Services server. An offline adapter contains the SQL queries for AIR SQLite for retrieving cached items like an assembler on the server retrieves items from the data source. An offline adapter can contain logic to perform fills based on criteria defined in a model just as the Model Assembler on the server contains logic to retrieve items from the data source based on criteria defined in a model.

This set of tasks builds an end-to-end model-driven application that works both online and offline. This application has an offline-enabled AIR desktop client that stores and manipulates data in an AIR SQLite database as well as the data source on the LiveCycle Data Services server.

When entities in a model contain annotations for offline code generation, offline adapter classes are automatically generated when you click the Generate Code button on the Modeler toolbar. The generated classes extend the `mx.data.SQLiteOfflineAdapter` class, which provides SQLite offline query capabilities for AIR clients that use data management. The generated classes override the `getQueryCriteria()`, `getQueryCriteriaParameters()`, and `getQueryOrder()` methods of the `SQLiteOfflineAdapter` class.

The overridden methods contain query code for each implicit filter and filter element in the model. Criteria-based filters, pass-through filters that contain simple JPQL queries and JPQL queries with cascading properties are supported. Pass-through filters that contain complex user-defined JPQL queries are not supported.

The following table describes the offline adapter methods.

Method	Description
<code>getQueryCriteria()</code>	Defines the SQL WHERE clause.
<code>getQueryCriteriaParameters()</code>	Defines the parameters of SQL WHERE clause.
<code>getQueryOrder()</code>	Defines the SQL ORDERBY clause.

This application uses the `lcds-samples` web application and the `ordersDB` data source on the LiveCycle Data Services server. These tasks are specific to a LiveCycle Data Services installation with an integrated Apache Tomcat application server. Application-server-specific configuration is noted where applicable.

Summary of steps

Perform the following tasks to build a model-driven offline application:

- 1 Install [LiveCycle Data Services](#), [Flash Builder](#), and the [application modeling plug-in](#). (Common with the *Build your first model-driven application* section.)

- 2 “Configure a JDBC data source” on page 327. (Common with the *Build your first model-driven application* section.)
- 3 “Configure RDS on the server” on page 328. (Common with the *Build your first model-driven application* section.)
- 4 “Configure server-side debug logging” on page 328. (Common with the *Build your first model-driven application* section.)
- 5 “Configure Flash Builder to use RDS” on page 329. (Common with the *Build your first model-driven application* section.)
- 6 “Create a Flex project in Flash Builder” on page 329.

Important: Set the application type of the Flex project to **Desktop** instead of Web. This type of offline-enabled application requires an AIR-based desktop client.

- 7 “Build a model and generate code” on page 334. (Specific to this workflow.)
- 8 “Create and run an offline-enabled desktop client” on page 335. (Specific to this workflow.)

Build a model and generate code

- 1 Start the samples database by running the startdb.bat or startdb.sh file, depending on your operating system. These files are located in the following directory:

install_root/sampledb

where *install_root* is the directory where you installed LiveCycle Data Services.

- 2 Start your application server if it isn't already running.
- 3 Start Flash Builder if it isn't already running.
- 4 In Flash Builder, make sure that you have created a Flex J2EE server project with Application Type set to **Desktop**, and it is the active project. This type of offline-enabled application requires an AIR-based desktop client.

Note: To make a project active, open an application file in the project and make it the active file in the editor.

- 5 In the Package Explorer, click the Filters button. In the Filters dialog, deselect ".model". This configuration lets you view model files in the Flash Builder Package Explorer.
- 6 In the Package Explorer, click the Open Model for Active Project button located at right end of the toolbar. Clicking this button for the first time creates a directory named .model and a model file with the same name as the Flex project. The model file opens automatically.

After the model is created, clicking the Open Model for Active Project button opens the model file for editing. The same button also appears in the toolbar for the Data/Services view when in the Flash perspective.

- 7 Switch to the Data Model perspective (the Modeler): click the Open Perspective button in the right corner of the Flash Builder window and then select Data Model.
- 8 Select the Design tab to enter the Design view.
- 9 By default, the RDS Data view is in the lower right side of the Data Model perspective. In the RDS Data view, expand the RDS server tree nodes to view the tables in the java:/comp/env/jdbc/ordersDB data source.
- 10 Select the PRODUCT table in the ordersDB data source and drag it to the Design view canvas. A Product entity is added to the model.
- 11 Select the Product entity in Design view and select the Code Gen tab in the Properties view below the Design view canvas.
- 12 In the Code Gen panel, click the Generate Offline Adapter checkbox and set the name of the Offline Adapter Package to com.adobe.offline.

13 Save the model and click the Deploy Model to LCDS server button. Accept the default settings and click OK.

Clicking OK generates a Model Assembler destination on the server for each entity. You can call these destinations from a Flex client to work with data stored in the database. By default, the model file is persisted in the WEB-INF/datamodel directory of the web application to which you deploy; the model is redeployed from this location on server startup.

14 Click the Generate Code button on the Modeler toolbar to generate the offline adapter code.

15 In the Package Explorer, expand the src folder, which now contains a com.adobe.offline package.

16 Expand the com.adobe.offline package, and double-click the ProductOfflineAdapter.as ActionScript class file to view it.

Note that the ProductOfflineAdapter class extends the mx.data.SQLiteOfflineAdapter class and overrides its `getQueryCriteria()`, `getQueryCriteriaParameters()`, and `getQueryOrder()` methods to provide query cases specific to the implicit filters in the Product entity: `getAll`, `getByProductId`, `getByDescription`, `getByPrice`, and `getByProductname`.

Create and run an offline-enabled desktop client

To use the client and server code generated from the model, create a Flex desktop (AIR) client that has a simple master-detail user interface.

Note: In Flash Builder, you must use a Flex project with Application Type set to **Desktop** rather than **Web**.

Complete the following steps to create a Flex desktop client:

- 1** Start the samples database if it isn't already running.
- 2** Start your application server if it isn't already running.
- 3** Start Flash Builder if it isn't already running.
- 4** In Flash Builder, switch to Design view in the Flash perspective.
- 5** In the Package Explorer, expand the nodes of your Flex project to src > (default package) and open the MXML file you want to use.
- 6** Set up a DataGrid control:
 - a** Select the Components tab in the lower left side of the Flash Builder window.
 - b** Expand the Controls folder in the Components tree.
 - c** Drag the DataGrid control to the Design view canvas.
 - d** Select the DataGrid control in the Design view canvas and set its editable property to true in the Properties view.
 - e** Select the Data/Services tab under the Design view canvas.
 - f** In the Data/Services view, expand the ProductService node.
 - g** Drag the `getAll () : Product []` function to the DataGrid control. Accept the default Bind To Data settings. The DataGrid columns changes to match the property names of the Product entity. T
- 7** Set up a Disconnect Button control:
 - a** In Design view, from the Controls folder in the Components tree, drag a Button control to the canvas below the DataGrid control.
 - b** Change the label property of the Button control to Disconnect and set its id property to disconnectBtn.
 - c** Right-click the Button control and select Generate Click Handler.

- d Modify the generated click handler code to match the following code. This click handler disconnects the serviceControl DataService instance from the server.

```
protected function disconnectBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.disconnect();
}
```

8 Set up a Connect Button control:

- a In Design view, from the Controls folder in the Components tree, drag a Button control to the canvas below the Disconnect Button control.
- b Change the label property of the Button control to Connect and set its id property to connectBtn.
- c Right-click the button control and select Generate Click Handler.
- d Modify the generated click handler code to match the following code. This click handler connects the serviceControl DataService to the server.

```
protected function connectBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.connect();
}
```

9 Set up a Save To Local Cache Button control:

- a In Design view, from the Controls folder in the Components tree, drag a Button control to the canvas to the right of the Disconnect Button control.
- b Change the label property of the Button control to Save To Local Cache and set its id property to saveCacheBtn.
- c Right-click the Button control and select Generate Click Handler.
- d Modify the generated click handler code to match the following code. This click handler saves data to the local SQLite database.

```
protected function saveCacheBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.saveCache();
}
```

10 Set up a Commit To Server Button control:

- a In Design view, from the Controls folder in the Components tree, drag a Button control to the canvas below the Save To Local Cache Button control.
- b Change the label property of the Button control to Commit To Server and set its id property to commitBtn.
- c Right-click the Button control and select Generate Click Handler.
- d Modify the generated click handler code to match the following code. This click handler commits data to the remote server-side database.

```
protected function commitBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.commit();
}
```

11 Set up Commit Required and Connected Label controls:

- a In Design view, from the Controls folder in the Components tree, drag a Label control to the canvas the right of the Save To Local Cache Button control.

- b Set the text property of the Label control to following binding code. This label displays the current value of the `serviceControl.commitRequired` property at run time.

```
{'Commit Required: ' + productService.serviceControl.commitRequired}
```

- c From the Controls folder in the Components tree, drag a Label control below the Commit Required Label control in the Design view canvas.

- d Set the text property of the Label control to the following binding code. This label displays the current value of the `serviceControl.connected` property at run time.

```
{'Connected: ' + productService.serviceControl.connected}
```

- 12 In Source view, add the following code to the script block of the MXML file. Replace all existing code that is above the event handlers for the DataGrid and Button controls.

```
import com.adobe.offline.ProductOfflineAdapter;
import mx.controls.Alert;
import mx.events.FlexEvent;
import mx.messaging.Channel;
import mx.messaging.ChannelSet;
import mx.messaging.channels.RTMPChannel;
import mx.messaging.events.ChannelEvent;
import mx.rpc.AsyncToken;
import mx.rpc.events.FaultEvent;
import mx.rpc.events.ResultEvent;

public var myOfflineAdapter:ProductOfflineAdapter;
public function channelConnectHandler(event:ChannelEvent):void
{
    productService.serviceControl.autoConnect=false;

}
protected function
    app_preinitializeHandler(event:FlexEvent):void
{
    var cs:ChannelSet = new ChannelSet();
    var customChannel:Channel = new RTMPChannel("my-rtmp",
        "rtmp://localhost:2037");
    cs.addChannel(customChannel);
    productService.serviceControl.channelSet=cs;
    customChannel.addEventListener(ChannelEvent.CONNECT,
        channelConnectHandler);
}
protected function
    windowedapplication1_creationCompleteHandler(event:FlexEvent):void
{
    productService.serviceControl.autoCommit = false;
    productService.serviceControl.autoConnect = false;
    productService.serviceControl.autoSaveCache = false;
    productService.serviceControl.offlineAdapter =
        new ProductOfflineAdapter();
    productService.serviceControl.fallBackToLocalFill=true;
    productService.serviceControl.encryptLocalCache = true;
    productService.serviceControl.cacheID = "myOfflineCache";
}
```

Especially for AIR applications, it is a best practice to create and assign channel sets at runtime. The code in the `app_preinitializeHandler()` method creates a channel set, adds a channel to it, and assigns the channel set to the `productService`.

The `windowedapplication1_creationCompleteHandler()` method sets the following properties that affect offline behavior:

Property	Description
<code>autoCommit</code>	Determines if the application automatically commits data changes.
<code>autoConnect</code>	Determines if the application automatically connects to the server when a connection is available.
<code>autoSaveCache</code>	Determines if data is automatically saved to the offline cache.
<code>offlineAdapter</code>	Assigns a custom offline adapter.
<code>fallBackToLocalFill</code>	Determines if the application falls back to using local fills from the AIR SQLite database when not connected to the server.
<code>encryptLocalCache</code>	Set this property to true to encrypt the SQLite database and ensure that data is unavailable to all other users and applications running on the same machine. The database is encrypted with a random encryption key and password using the SQLite encryption feature. The password is stored locally using AIR EncryptedLocalStore support, associated with the database name.
<code>cacheID</code>	Assigns a cache ID for storing the local copy of data in the AIR SQLite database.

13 Add the following event handler references to the `WindowedApplication` element:

```
preinitialize="app_preinitializeHandler(event)"  
creationComplete="windowedapplication1_creationCompleteHandler(event)"
```

14 Return to Design view.

15 Make sure you have server-side logging configured as described in “[Configure server-side debug logging](#)” on page 328.

Position the AIR window and the application server console side by side.

16 Run the application.

The DataGrid control fills with data; the `productService.getAll()` function is called in the DataGrid control’s `creationComplete` event handler.

Note: Calling the `getAll()` function results in a call to the `fill()` method of the underlying `DataService` instance.

17 Change the data in one or more DataGrid cell.

18 Click Commit. The client calls the `serviceControl.commit()` method to submit the changed data to the server database.

19 Make additional data changes and click Save To Local Cache. The client calls the `serviceControl.saveCache()` method to submit data changes to the local SQLite database.

20 Stop the application server.

21 Close the client and then run it again.

The DataGrid control fills with data from the local SQLite database.

Calling the `getAll()` function still resulted in a call to the `fill()` method of the underlying DataService instance. However, that `fill()` call performed the local `getAll` query specified in the ProductOfflineAdapter class because the `DataService.fallBackToLocalFill` property is set to `true`.

22 Restart the application server. Click Commit To Server and watch the local changes get persisted to the server database.

Similarly, you can make data changes, and then click Disconnect, Save To Local Cache, Reconnect, and Commit To Server.

Note: This application does not use a model-driven form because of explicit calls to the `DataService.commit()` method in the generated model-driven form MXML component. Those call result in connection errors when you click the model-driven form buttons while offline.

Completed MXML file

The following example shows the completed MXML file for the offline-enabled client:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
xmlns:mx="library://ns.adobe.com/flex/mx" xmlns:OfflineAIRAPP="OfflineAIRAPP.*"
preinitialize="app_preinitializeHandler(event)"
creationComplete="windowedapplication1_creationCompleteHandler(event)">
    <fx:Script>
        <![CDATA[
            import com.adobe.offline.ProductOfflineAdapter;

            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.messaging.Channel;
            import mx.messaging.ChannelSet;
            import mx.messaging.channels.RTMPChannel;
            import mx.messaging.events.ChannelEvent;
            import mx.rpc.AsyncToken;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            public var myOfflineAdapter:ProductOfflineAdapter;
            public function channelConnectHandler(event:ChannelEvent):void
            {
                productService.serviceControl.autoConnect=false;

            }
            protected function
                app_preinitializeHandler(event:FlexEvent):void
            {
                var cs:ChannelSet = new ChannelSet();
                var customChannel:Channel = new RTMPChannel("my-rtmp",
                    "rtmp://localhost:2037");
                cs.addChannel(customChannel);
                productService.serviceControl.channelSet=cs;
                customChannel.addEventListener(ChannelEvent.CONNECT,
                    channelConnectHandler);
            }
        ]]>
    </fx:Script>

```

```
}

protected function dataGrid_creationCompleteHandler(event:FlexEvent):void
{
    getAllResult.token = productService.getAll();
}

protected function
    windowedapplication1_creationCompleteHandler(event:FlexEvent):void
{
    productService.serviceControl.autoCommit = false;
    productService.serviceControl.autoConnect = false;
    productService.serviceControl.autoSaveCache = false;
    productService.serviceControl.offlineAdapter = new
        ProductOfflineAdapter();
    productService.serviceControl.fallBackToLocalFill=true;
    productService.serviceControl.encryptLocalCache = true;
    productService.serviceControl.cacheID = "myOfflineCache";
}

protected function connectBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.connect();
}

protected function DisconnectBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.disconnect();
}

protected function commitBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.commit();
}

protected function saveCacheBtn_clickHandler(event:MouseEvent):void
{
    productService.serviceControl.saveCache();
}

]]>
</fx:Script>
<fx:Declarations>
    <s:CallResponder id="getAllResult" />
    <OfflineAIRAPP:ProductService id="productService"
        fault="Alert.show(event.fault.faultString + '\n' +
        event.fault.faultDetail)"/>
</fx:Declarations>
<mx:DataGrid editable="true" x="9" y="10" id="dataGrid"
    creationComplete="dataGrid_creationCompleteHandler(event)"
    dataProvider="{getAllResult.lastResult}">
```

```
<mx:columns>
    <mx:DataGridColumn headerText="productid" dataField="productid"/>
    <mx:DataGridColumn headerText="description" dataField="description"/>
    <mx:DataGridColumn headerText="price" dataField="price"/>
    <mx:DataGridColumn headerText="productname" dataField="productname"/>
</mx:columns>
</mx:DataGrid>
<s:Button x="10" y="246" label="Connect" click="connectBtn_clickHandler(event)" id="connectBtn" width="84" height="30"/>
<s:Button x="112" y="204" label="Save to Local Cache" id="saveCacheBtn" click="saveCacheBtn_clickHandler(event)" height="30"/>
<s:Button x="110" y="246" label="Commit to Server" id="commitBtn" click="commitBtn_clickHandler(event)" width="135" height="30"/>
<s:Button x="10" y="204" label="Disconnect" id="DisconnectBtn" click="DisconnectBtn_clickHandler(event)" height="30"/>
<s:Label x="270" y="204" text="{'Commit Required: ' + productService.serviceControl.commitRequired}"/>
<s:Label x="270" y="246" text="{'Connected: ' + productService.serviceControl.connected}"/>
</s:WindowedApplication>
```

More Help topics

[“Building the client application” on page 372](#)

Customizing client-side functionality

You can customize client-side functionality in the following ways when using Adobe modeling technology with Flash Builder:

- Modify generated ActionScript service wrapper and value object source files
- Modify the template files from which the ActionScript source files and the model-driven form are generated

Note: The ability to modify the templates for ActionScript service wrapper and value object source files was introduced in LiveCycle Data Services 3.1.

Generated ActionScript source files

The ActionScript source files are based on a model. Service wrappers and value objects are generated for every entity in the model; there is an implicit service for every entity in a model. Service wrappers are also generated for every explicit service element in the model. Each generated wrapper and value object has a superclass that you do not change and a subclass you can change. The subclasses do not get regenerated when you regenerate ActionScript code for the application, so your changes are preserved.

The following table describes generated service wrapper and value object source files:

Class file	Description
_Super_ServiceName.as	The parent class of the service class that is overwritten for each generation. This class contains the service functions.
ServiceName.as	The service class in which you can implement custom functionality by overriding the superclass behavior.
_Super_EntityName.as	The parent class of the value object that is overwritten for each generation. This class contains the properties and methods that an entity defines.
EntityName.as	The value object class in which you can implement custom functionality by overriding the superclass behavior.
EntityNameEntityMetadata.as	Contains information about an entity and its relationship with other entities in the model.

To build an application that uses a custom method, see “[Implement and use a client-side entity method](#)” on page 342.

Code generation templates

The service wrapper and value object files are generated based on a code generation template. For more information, see “[Customizing client-side code generation](#)” on page 345.

Implement and use a client-side entity method

This set of tasks walks through the process of building an end-to-end model-driven application that uses an entity method implemented in an ActionScript value object class.

Assume that you create a model that is based on a relational database. Next, assume that you want to calculate a discount price based on a product price from the database. To meet your business needs, you can implement a custom method in a value object that calculates the discount price. After you modify the generated Product value object, your new application logic becomes part of the client-side functionality.

You perform similar steps to implement a custom service function in a generated service wrapper.

To implement the entity method, you perform these steps:

- 1 Add a discount method to the Product entity in the model to calculate a discount price based on the price value from the database.
- 2 Add a discountPrice derived property to the Product entity in the model and set its expr (expression) value to call the discount method.
- 3 Regenerate the ActionScript source files.
- 4 Implement a discountPrice() method in the Product value object that is generated from the Product entity in the model. The superclass of the Product class contains a stub for the discountPrice() method with the correct signature. The implementation in the Product value object overrides that stub method.

Summary of steps

Perform the following tasks to build a model-driven application that uses a client-side value object method:

- 1 Install [LiveCycle Data Services](#), [Flash Builder](#), and the [application modeling plug-in](#). (Common with the *Build your first model-driven application* section.)
- 2 “[Configure a JDBC data source](#)” on page 327. (Common with the *Build your first model-driven application* section.)
- 3 “[Configure RDS on the server](#)” on page 328. (Common with the *Build your first model-driven application* section.)

- 4 “Configure server-side debug logging” on page 328. (Common with the *Build your first model-driven application* section.)
- 5 “Configure Flash Builder to use RDS” on page 329. (Common with the *Build your first model-driven application* section.)
- 6 “Create a Flex project in Flash Builder” on page 329. (Common with the *Build your first model-driven application* section.)
- 7 “Build a model and generate code” on page 343. (Specific to this workflow.)
- 8 “Create and run a Flex client that uses a custom method” on page 344. (Specific to this workflow.)

Build a model and generate code

- 1 Start the samples database by running the startdb.bat or startdb.sh file, depending on your operating system. These files are located in the following directory:

install_root/sampledb

where *install_root* is the directory where you installed LiveCycle Data Services.

- 2 Start your application server if it isn't already running.
- 3 Start Flash Builder if it isn't already running.
- 4 In Flash Builder, make sure that you have created a Flex J2EE server project and it is the active project.
Note: To make a project active, open an application file in the project and make it the active file in the editor.
- 5 In the Package Explorer, click the Filters button. In the Filters dialog, deselect ".model". This configuration lets you view model files in the Flash Builder Package Explorer.
- 6 In the Package Explorer, click the Open Model for Active Project button located at right end of the toolbar. Clicking this button for the first time creates a directory named .model and a model file with the same name as the Flex project. The model file opens automatically.

After the model is created, clicking the Open Model for Active Project button opens the model file for editing. The same button also appears in the toolbar for the Data/Services view when in the Flash perspective.

- 7 Switch to the Data Model perspective (the Modeler): click the Open Perspective button in the right corner of the Flash Builder window and then select Data Model.
- 8 Select the Design tab to enter the Design view.
- 9 By default, the RDS Data view is in the lower right side of the Data Model perspective. In the RDS Data view, expand the RDS server tree nodes to view the tables in the java:/comp/env/jdbc/ordersDB data source.
- 10 Select the PRODUCT table in the ordersDB data source and drag it to the Design view canvas. A Product entity is added to the model based on the database table. The database table consists of the following fields:
 - **ProductId**: Stores the identifier value of the item.
 - **Description**: Stores a description of the item.
 - **Price**: Stores the price of the item.
 - **ProductName**: Store the name of the item.
- 11 Switch to Source view, and add the code shown in bold to the Product entity:

```
<entity name="Product" persistent="true">
...
<property name="discountPrice" type="float" expr="discount(price)"/><method
name="discount" arguments="p:float" return-type="float"/>
</entity>
```

The discountPrice property is a derived property that calls the discount method in its expression.

12 Save the model file to regenerate the ActionScript code.

13 Click the Deploy Model to LCDS server button. Accept the default settings and click OK.

Clicking OK generates a Model Assembler destination on the server for each entity in the model. You can call these destinations from a Flex client to work with data stored in the database. By default, the model file is saved in the WEB-INF/datamodel directory of the web application to which you deploy.

14 In the Package Explorer, expand the src folder.

15 Expand the *ProjectName* package, where *ProjectName* is the name of your Flash Builder project.

16 Double-click the _SuperProduct.as ActionScript class file to view it.

Note that the _SuperProduct class contains a `discount()` method stub generated from the discount method element in the model.

17 Double-click the Product.as ActionScript class file to view it.

18 In the Product.as file, add the following method code below the `END OF DO NOT MODIFY SECTION` comment:

```
override public function discount (p:Number):Number
{
    var discountPrice:Number = p - (p * .05);
    return discountPrice;
}
```

19 Save the Product.as file. This customized file is not overwritten when you regenerate code for the project. Only the superclass is overwritten; your customization is preserved.

Create and run a Flex client that uses a custom method

To use the client and server code generated from the model, create a Flex web client that has a simple master-detail user interface. For detailed information, see “[Building the client application](#)” on page 372.

Complete the following steps to create a Flex web client:

- 1** Start the samples database if it isn't already running.
- 2** Start your application server if it isn't already running.
- 3** Start Flash Builder if it isn't already running.
- 4** In Flash Builder, switch to Design view in the Flash perspective.
- 5** In the Package Explorer, expand the nodes of your Flex project to src > (default package) and open the displayed MXML file.
- 6** Select the Components tab in the lower left side of the Flash Builder window.
- 7** Expand the Controls folder in the Components tree.
- 8** Drag the DataGrid control to the Design view canvas.
- 9** Select the Data/Services tab under the Design view canvas.
- 10** In the Data/Services view, expand the ProductService node.

- 11 Drag the `getAll() : Product[]` function to the DataGrid control. The DataGrid columns changes to match the property names of the Product entity.
- 12 Right click the DataGrid control and select Generate Details Form.
- 13 Select Model Driven Form and click OK to generate a Form.
- 14 Reposition the form so that it is not covering the DataGrid control.
- 15 Run the application. The DataGrid control fills with data; the `productService.getAll()` function is called in the DataGrid control's creationComplete event handler.

The values of the `discountPrice` property in the DataGrid control and the form automatically display the value that the `Product.discount()` method returns.

Note: Calling the `getAll()` function results in a call to the `fill()` method of the underlying DataService instance.

Customizing client-side code generation

Service wrappers, value objects, offline adapters, and model-driven forms are generated from FreeMarker templates. FreeMarker is a Java-based template engine; for more information see freemarker.org.

You can generate and customize these templates to change the structure of the generated files. When you generate templates, Flash Builder uses the generated templates instead of the default templates.

To extract templates for a Flash Builder project:

- 1 In the Flash Builder Package Explorer, right click your model file.
- 2 From the pop-up menu, select Data Model > Extract Templates For Active Generators to generate template files.

Note: You are prompted to replace an existing template file before it is overwritten.

You can change the location in which the templates are extracted in the Code Generation Preferences dialog. To change the template locations:

- 1 Select Window > Preferences > Adobe > Data Model > Code Generation.
- 2 In the Code Generation dialog, select the Generator Configuration button for ActionScript Generation or Model Driven Form and change the Template Lookup Path field.
- 3 Click OK.

The template files are searched for in the location specified in the Template Lookup Path. If they are not found in this location, then default values are used. The lookup directory is where the extract operation places the files. The following illustration shows the Template Lookup Path field.

Note: There is no guarantee that your customized template and generated code will work in future releases of LiveCycle Data Services. Also, to apply a LiveCycle Data Services patch or update, you must extract the templates again.

Service wrapper templates

You can generate and customize the following template files for service wrapper classes:

Template file	Description
ASCustomServiceChild.ftl	Generates <code>ServiceName.as</code> for custom service
ASCustomServiceSuper.ftl	Generates <code>_Super_ServiceName.as</code> for custom service
ASDataManagementServiceChild.ftl	Generates <code>ServiceName.as</code> for Data Management Service wrapper

Template file	Description
ASDataManagementServiceSuper.ftl	Generates _Super_ServiceName.as for Data Management Service wrapper
ASRPCServiceChild.ftl	Generates ServiceName.as for RPC service wrapper
ASRPCServiceSuper.ftl	Generates _Super_ServiceName.as for RPC service wrapper

By default, the templates are generated in the templates/as directory of the Flash Builder project.

The templates use the following tokens that correspond to Java classes to perform code generation. See the LiveCycle Data Services Javadoc documentation for more information about the corresponding Java APIs.

Template token	Description
model	Corresponds to fiber.core.api.Model
service	Corresponds to fiber.core.api.Service
util	Corresponds to fiber.gen.as.utils.ASGenerationUtil
packageMap	Corresponds to Map of String to String: entity names to package names
typeHelper	Corresponds to fiber.gen.as.wrapper.ASServiceGenTypeHelper
termFormatter	Corresponds to fiber.gen.as.ASTermFormatter

To better understand the templates, see the related Javadoc, examine the source code of the generated classes, and see *Client code generation* in the [Application Modeling Technology Reference](#).

For information about the related ActionScript APIs, see:

- com.adobe.fiber.valueobjects
- com.adobe.fiber.services.wrapper.DataServiceWrapper

Value object templates

You can generate and customize the following templates for value object classes:

Template file	Description
ASValueObjectChild.ftl	Generates EntityName.as
ASValueObjectMetaData.ftl	Generates _EntityNameEntityMetadata.as
ASValueObjectSuper.ftl	Generates _Super_EntityName.as

By default, the templates are generated in the templates/as directory of the Flash Builder project.

The templates use the following tokens that correspond to Java classes to perform code generation. See the LiveCycle Data Services Javadoc documentation for more information about the corresponding Java APIs.

Template token	Description
model	Corresponds to fiber.core.api.Model
entity	Corresponds to fiber.core.api.Entity
util	Corresponds to fiber.gen.as.utils.ASGenerationUtil

Template token	Description
packageMap	Corresponds to Map of String to String: entity names to package names
typeHelper	Corresponds to fiber.gen.as.wrapper.ASServiceGenTypeHelper
termFormatter	Corresponds to fiber.gen.as.ASTermFormatter

To better understand the templates, see the related Javadoc, examine the source code of the generated classes, and see *Client code generation* in the [Application Modeling Technology Reference](#).

For information about the related ActionScript APIs, see:

- com.adobe.fiber.valueobjects
- com.adobe.fiber.services.wrapper.DataServiceWrapper

Offline adapter template

You can generate and customize the following template for offline adapter classes:

Template file	Description
ASOfflineAdapter.ftl	Generates <i>EntityNameOfflineAdapter.as</i>

By default, the template is generated in the templates/as directory of the Flash Builder project.

The template uses the following tokens that correspond to Java classes to perform code generation. See the LiveCycle Data Services Javadoc documentation for more information about the corresponding Java APIs.

Template token	Description
offlineHelper	Corresponds to fiber.gen.as.offline.OfflineHelper
entity	Corresponds to fiber.core.api.Entity
util	Corresponds to fiber.gen.as.utils.ASGenerationUtil
packageMap	Corresponds to Map of String to String: entity names to package names
typeHelper	Corresponds to fiber.gen.as.wrapper.ASServiceGenTypeHelper
termFormatter	Corresponds to fiber.gen.as.ASTermFormatter

To better understand the templates, see the related Javadoc, examine the source code of the generated classes.

Model-driven form template

You can change the appearance of the model-driven form by customizing the following template:

Template file	Description
ASModelDrivenForm.ftl	Generates <i>EntityNameForm.mxml</i> and associated images

By default, the template is generated in the templates/as/form directory of the Flash Builder project. After modifying the template, regenerate the form in Flash Builder to apply your changes.

Typically, you modify the appearance of Flex components or change the location of components in the model-driven form. For example, you can use standard Button images or a List control in place of a ComboBox control. The two inputs to the model-driven form are the service wrapper and value object (data type) that are generated from an entity in the model.

You can change the FreeMarker expressions that determine what code is generated around the service wrapper and value object. However, this is an advanced task that requires knowledge of the Java and ActionScript modeling APIs. The template uses the following tokens that correspond to Java classes to perform code generation. See the LiveCycle Data Services Javadoc documentation for more information about the corresponding Java APIs.

Template token	Description
model	Corresponds to fiber.core.api.Model
entity	Corresponds to fiber.core.api.Entity
util	Corresponds to fiber.gen.as.utils.ASGenerationUtil
packageMap	Corresponds to Map of String to String: entity names to package names
typeHelper	Corresponds to fiber.gen.as.wrapper.ASServiceGenTypeHelper
termFormatter	Corresponds to fiber.gen.as.ASTermFormatter

To better understand the templates, see the related Javadoc, examine the source code of the generated MXML component and the service wrapper and value object classes, and see *Client code generation* in the [Application Modeling Technology Reference](#).

For information about the related ActionScript APIs, see:

- com.adobe.fiber.valueobjects
- com.adobe.fiber.services.wrapper.DataServiceWrapper

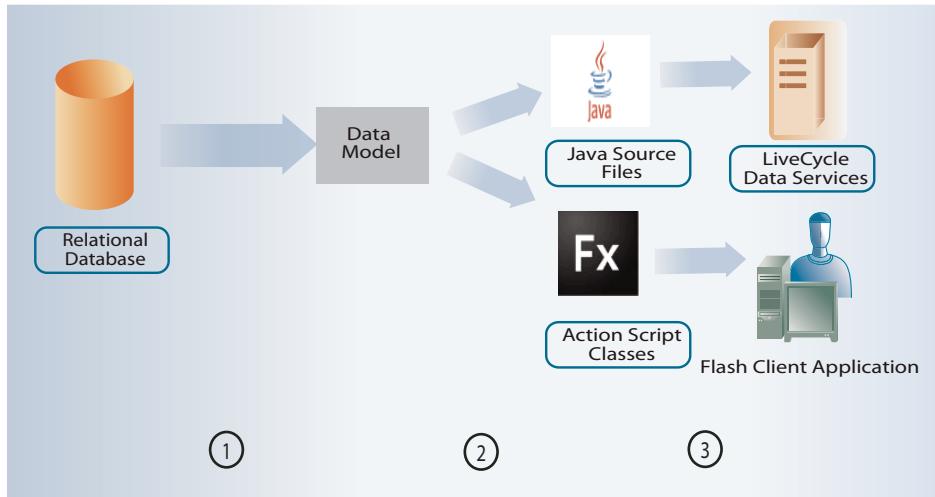
Customizing server-side functionality

You can customize server-side functionality by modifying Java source files that are created when using Adobe modeling technology. You can configure Flash Builder to create Java source files.

The Java source files are based on a model. Assume, for example, that you create a model that is based on a relational database. Next assume that you want to implement application logic to track the date and time that a record was deleted. You can modify Java source files to meet your business requirements. After you deploy the Java source files to the J2EE application server, your application logic becomes part of the server-side functionality.

Note: The ability to generate Java source files that are based on a model was introduced in LiveCycle Data Services 3.1.

The following illustration provides a visual representation of Java source files being created by using Adobe modeling technology.



The Customizing server-side functionality section discusses how to modify Java source files generated. For information about modifying the ActionScript classes that are generated from a data model, see “[Customizing client-side functionality](#)” on page 341.

The following table describes the steps in this illustration.

Step	Description
1	A data model is created based an existing relational database.
2	Java source files and ActionScript classes are created based on the data model.
3	The Java source files are modified and deployed to the J2EE application server hosting LiveCycle Data Services. Adobe modeling technology also creates ActionScript source files that you can use to develop a client application.

Note: You can use any valid data model to generate Java source files, not just a data model based on a relational database. The data model used in Customizing server-side functionality is based on a relational database.

Generated Java source files

The following Java source files are created for each entity in the data model:

- **_Super_<entity>.java:** The parent class that is overwritten for each generation. This class contains each identifier value and data property in the entity as a member along with getter and setter functions. This class also contains methods that an entity defines. The method contains implementation logic. In addition, both equals and hashCode methods are created by using the identifier properties as the definition of equals. For example, objects with the same id fields are equal. This class and its data properties are JPA annotated for use in the persistence layer. Do not modify a _Super_<entity>.java file.
- **<entity>.java:** This class extends the super class, and is generated only once. It is an empty class that can be customized with business logic. This class does contain at least one JPA annotation, @Entity, and possibly the @IdClass and @Table annotations. This class is only generated if it does not exist.

- **<entity>PK.java:** This class is generated when an entity has more than one id property. This class is also generated when the id property is a persistent entity with more than one id property. The persistence layer for the entity uses the Primary Key (PK) class. This class is JPA annotated for use in the persistence layer.
- **<entity>BeanInfo.java:** In the event of one or more properties in an entity with a name that does not conform to the JavaBean standard, a BeanInfo class is generated for the entity class. This class provides the correct getter and setter function names for the JavaBean API used by various parts of the service side logic.
- **<entity>Assembler.java:** This class extends the FiberAssembler class and is generated once. This class contains a subset of the methods defined by the AbstractAssembler class which all delegate to the super class. This class is provided as a base class for customization. It is not generated by default.

Note: The example in *Customizing server-side functionality* modifies the Assembler.java class. The application logic is added to the deleteItem() method and tracks the date and time that a record was deleted.

Implementing an assembler method on the server

You can customize the Assembler.java class that is generated based on a data model. The data model created in *Implementing an assembler method on the server* is based on the Product table. This table is located in the sample database that is available with LiveCycle Data Services. The Product table consists of the following fields:

- **Products:** Stores the identifier value of the item.
- **Description:** Stores a description of the item.
- **Price:** Stores the price of the item.
- **ProductName:** Store the name of the item.

If you generate Java source files that are based on a data model that references the Product table, the following methods are located in the ProductAssembler.java file. This assembler extends fiber.data.assemblers.FiberAssembler, which extends flex.data.assemblers.AbstractAssembler.

Methods inherited AbstractAssembler:

- `count()`: Retrieve the number of items for a given query.
- `createItem()`: Creates an item (a new record).
- `deleteItem()`: Deletes an item (an existing record).
- `fill()`: Performs a fill operation.
- `getItem()`: Retrieves an item with the specified map of identifier values.
- `updateItem()`: Updates a specific item (a record).

Methods specific to FiberAssembler:

- `validateCreateItem()`: Called by the `createItem()` method just before delegating to the persistence layer.
- `validateDeleteItem()`: Called by the `deleteItem()` method just before delegating to the persistence layer.
- `validateUpdateItem()`: Called by the `updateItem()` method just before delegating to the persistence layer.

You can customize an assembler method on the server to meet your business requirements. For example, you can create application logic in the `deleteItem()` method that tracks the date and time when an item is deleted. In addition, you can create application logic that prevents certain records from being deleted.

To customize the Assembler class as part of creating a client application, perform the following tasks:

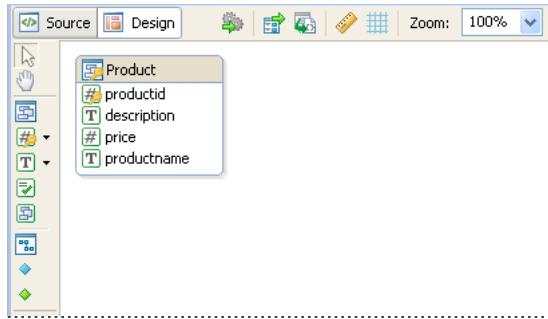
- 1 Install [LiveCycle Data Services](#), [Flash Builder](#), and the [application modeling plugin](#). (Common with the *Build your first model-driven application* section.)

- 2 “Configure a JDBC data source” on page 327. (Common with the *Build your first model-driven application* section.)
- 3 “Configure RDS on the server” on page 328. (Common with the *Build your first model-driven application* section.)
- 4 “Configure server-side debug logging” on page 328. (Common with the *Build your first model-driven application* section.)
- 5 “Configure Flash Builder to use RDS” on page 329. (Common with the *Build your first model-driven application* section.)
- 6 “Create a Flex project in Flash Builder” on page 329. (Common with the *Build your first model-driven application* section.)
- 7 “Create a data model based on the Product table” on page 351. (Specific to this workflow.)
- 8 “Generate Java source files based on the data model” on page 353. (Specific to this workflow.)
- 9 “Generate Java source files based on the data model” on page 353. (Specific to this workflow.)
- 10 “Modify the Java source files” on page 353. (Specific to this workflow.)
- 11 “Deploy the Java source files to the J2EE application server” on page 356. (Specific to this workflow.)
- 12 “Modify the data model” on page 356. (Specific to this workflow.)
- 13 “Create and run a Flex client” on page 357. (Specific to this workflow.)

Note: The remaining sections discuss each of the steps marked as *specific to this workflow* in detail.

Create a data model based on the Product table

You can create a data model that is based on a relational database after the RDS server is configured. A data model is created in Flash Builder. The example data model is based on a single table named Product. The data model is an FML file that contains XML elements. After you create a data model, the FML file becomes part of your project. The following illustration shows a visual representation of the Product data model in the Data Model perspective.



The following code example represents the corresponding Product.fml file.

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <annotation name="DMS">
        <item name="datasource">java:/comp/env/jdbc/ordersDB</item>
        <item name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</item>
    </annotation>
    <entity name="Product" persistent="true">
        <annotation name="ServerProperties">
            <item name="ServerType">LCDS</item>
        </annotation>
        <annotation name="DMS">
            <item name="Table">PRODUCT</item>
        </annotation>
        <id name="productid" type="integer">
            <annotation name="DMS">
                <item name="ColumnName">PRODUCTID</item>
            </annotation>
        </id>
        <property name="description" type="string" length="255">
            <annotation name="DMS">
                <item name="ColumnName">DESCRIPTION</item>
            </annotation>
        </property>
        <property name="price" type="float">
            <annotation name="DMS">
                <item name="ColumnName">PRICE</item>
            </annotation>
        </property>
        <property name="productname" type="string" length="255">
            <annotation name="DMS">
                <item name="ColumnName">PRODUCTNAME</item>
            </annotation>
        </property>
    </entity>
</model>
```

- 1 From within the Package Explorer, right click your project and select New > Other.
- 2 In the New dialog, open the Data Model option.
- 3 Select Data Model and click Next.
- 4 In the New Data Model dialog, select your project.
- 5 In the Filename field, specify the filename for your model. Name your data model Product. Click Next.
- 6 Select the From A Database option. Click Next.
- 7 From the RDS Server drop-down list, select the RDS server that you configured.
- 8 From the Data Source drop-down, select the data source that you configured. Select java:/comp/env/jdbc/ordersDB.
- 9 In the Table field, select the tables. Select Product.
- 10 Click Finish.
- 11 In the Package Explorer, select the FML file that represents your data model.
- 12 Right click and from the pop-up menu, select Data Model > Deploy Data Model To The LCDS Server.
- 13 In the Deploy Data Model dialog, select the server from the Server list.
- 14 In the Deploy Data Model dialog, keep the default data model name. Leave the overwrite option deselected.

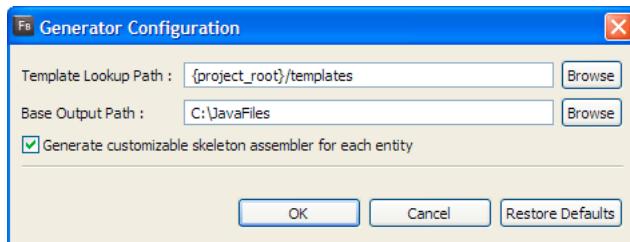
15 Select the Unchanged option for the Database Table options.

16 Click Finish. A confirmation message appears if the data model is successfully deployed to the J2EE application server.

Note: When you redeploy a model to the server with the Update radio button selected, verify that your database tables are updated correctly. There are situations where Hibernate does not drop table columns even though you have removed the corresponding properties from entities in the model. This happens when a database owner (DBO) name is a prefix in the table name; for example, PUBLIC.EMPLOYEE, where PUBLIC is the DBO name. There is no indication from Hibernate or the Modeler that the update failed. You can work around this issue by removing the DBO name from the table name. When looking for a table, Hibernate prepends the DBO (PUBLIC, in this case) to the table name. If the table name in the model is already PUBLIC.EMPLOYEE, Hibernate looks for PUBLIC.PUBLIC.EMPLOYEE and fails to find it.

Generate Java source files based on the data model

After you create the data model, you can generate Java source files that are based on the data model. To generate Java source files, you configure the Java Generator Configuration option within Flash Builder, as shown in the following illustration.



In the Base Output Path field, specify the location where the Java source files are created. In this example, the Java source files are written to the C:\JavaFiles directory. The Java source files are placed within a package name that matches the data model name. For example, if the data model is named Product, then the package name is Product.

Click the Generate customizable skeleton assembler for each entity option to create an Assembler.java file. This file contains methods that perform operations on the data. For example, this class contains a `deleteItem()` method that can be customized.

- 1 From the Flash Builder menu, select Window > Preferences.
- 2 In the Preferences dialog, select Adobe > Data Model > Code Generation.
- 3 Select Default from the Java drop-down list to enable Java generation. (Java generation is disabled by default.)
- 4 Click Configure Generator.
- 5 Specify the directory location in which to write the generated Java source files.
- 6 Click the Generate Customizable Skeleton Assembler For Each Entity option to create an Assembler.java file.
- 7 Click OK.
- 8 Change the perspective in Flash Builder to Data Model.
- 9 From the toolbar above the Source view, click the Generate Code button.

Modify the Java source files

After you create the Java source files, you can modify them to meet your business requirements. For example, assume that you want to track the date and time when a record was deleted. In this situation, add Java application logic to the `deleteItem()` method that is located in the ProductAssembler.java file.

To retrieve an item's data values, create a Product instance that represents the deleted record by casting `item` to Product. Then you can invoke a method to retrieve a data value. For example, you can invoke the `getProductname()` method to get the item name, as the following example shows.

```
//Get the name and price of the item being deleted
Product myProduct = (Product)item;
Float myPrice = myProduct.getPrice();
String strPrice = myPrice.toString();
String strName = myProduct.getProductname();
```

The `item` object is a parameter of the `deleteItem()` method. You can only cast `item` to a strongly typed data type if the cast value is based on a generated class that is deployed to LiveCycle Data Services. In this example, `Product` represents the `Product.java` file that is generated and deployed to LiveCycle Data Services.

Likewise, you can retrieve the value of an item by using a `PropertyProxy` instance. Use the `BeanProxy` constructor and pass `item`. Then retrieve the value by invoking the `PropertyProxy` instance's `getValue()` method and pass a string value that specifies the value to retrieve.

```
PropertyProxy p = new BeanProxy(item);
Float f = (Float) p.getValue("price");
```

You can also control whether to delete an item based on conditions such as a data value. For example, assume that you do not want to delete a record where the item is a memory stick. You can add application logic to prevent a record being deleted. The following example shows the modified `deleteItem()` method that tracks the date and time when an item is deleted. If the user attempts to delete a record where an item is a memory stick, an exception is thrown.

```
public void deleteItem(Object item)
{
    try
    {

        //Get the description and price of the item being deleted
        Product myProduct = (Product)item;
        Float myPrice = myProduct.getPrice();
        String strPrice = myPrice.toString();
        String strName = myProduct.getProductname();

        File myFile = new File("C:\\log.txt");
        BufferedWriter out = new BufferedWriter(new FileWriter(myFile));

        //Do not allow flash memory to be deleted
        if (strName.equals("memory stick"))
        {
            //Write a message to the log file that flash memory cannot be deleted
            out.write("A record where item is memory stick cannot be deleted");
            out.close();

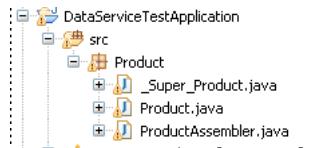
            //Throw an Exception
            Exception myException = new Exception("A record where item
                is memory stick cannot be deleted.");
        }
    }
}
```

```
        throw myException;
    }

    //Get the time when the item is deleted
    String DATE_FORMAT_NOW = "yyyy-MM-dd HH:mm:ss";
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat(DATE_FORMAT_NOW);
    String myTime = sdf.format(cal.getTime());

    //Write out the values to the log file
    out.write("The "+strName+ " item priced at "+ strPrice +
              " was deleted at: "+myTime);
    out.close();
    super.deleteItem(item);
}
catch (Exception e)
{
    e.printStackTrace();
}
}
```

To modify Java source files, copy the Java files from the location specified by using the Generator Configuration option to a Java IDE. When you copy the Java source files to your Java project, place the Java files within the correct package. By default, the name of the package matches the name of the data model. The following illustration shows a Java project with the Java source files. Notice that the package is named Product.



Add the LiveCycle Data Service JAR files to your project's class path. The LiveCycle Data Service JAR files are located in the following path:

[Install Directory]/lcds/tomcat/webapps/lcds-samples/WEB-INF/lib

where [Install Directory] is the LiveCycle Data Services installation location. Ensure that you add the following JAR files to the class path:

- flex-messaging-core.jar
- flex-messaging-common.jar
- flex-messaging-data.jar
- hibernate-annotation.jar
- ejb3-persistence.jar
- fiber-lcds.jar

Within the Java IDE, you can modify the Java files to meet your business requirements. After you are done modifying the Java source files, you can compile them.

Note: Instead of modifying the generated Assembler class, you can create your own Assembler class that extends `FiberAssembler`. (See “[Creating a Java class that extends a LiveCycle Data Services class](#)” on page 21.)

Deploy the Java source files to the J2EE application server

After you modify and compile the Java source files, package them into a JAR file. For example, if you are using Eclipse, export your Java project as a JAR file. Place the JAR file in the application's lib directory. Assuming that the application is named lcds-samples, place the JAR file into the following directory:

```
[Install Directory]/lcds/tomcat/webapps/lcds-samples/WEB-INF/lib
```

where [Install Directory] is the LiveCycle Data Services installation location.

Modify the data model

After the JAR file is deployed to the J2EE application server hosting LiveCycle Data Services, modify the data model to reference the modified Java classes. Ensure that your ProductAssembler Java class is used. To modify the data model, use the Modeler in Flash Builder. Under Data Mgmt, add the value Product.ProductAssembler to the Assembler Class option.

After you modify the data model to reference the ProductAssembler class, redeploy the data model to the server. For more information about using the Modeler in Flash Builder, see the [see the Application Modeling Technology Reference](#).

- 1 Change the perspective to Data Model in Flash Builder. Make sure the view is in Design view.
- 2 Select the Product data model.
- 3 In the Properties view, select the Data Mgmt tab.
- 4 Change the Assembler Class option to Product.ProductAssembler.
- 5 Save the data model.
- 6 In Project Explorer, right click your data model file. From the pop-up menu, select Data Model > Deploy Data Model To The LCDS Server.
- 7 In the Deploy Data Model dialog, select the server from the Server list.
- 8 In the Deploy Data Model dialog, keep the default data model name.
- 9 Select the overwrite option.
- 10 Click Finish. A confirmation message appears if the data model is successfully deployed to the J2EE application server.

Note: If you do not deploy the JAR file that contains the Java classes, you are not able to redeploy the data model. This is because the data model references the Java classes and cannot be deployed if the Java classes are not deployed.

Create and run a Flex client

Create a client application in Flash Builder that is based on the data model. When you delete an item, the Java code located in the assembler's `deleteItem()` method tracks the date and time that the item was deleted.

The screenshot shows a Flex application interface. At the top is a DataGrid control displaying five rows of product data. Below the DataGrid is a form containing three text input fields: 'Description' (containing '100 cds'), 'Price' (containing '22'), and 'Productname' (containing 'case of cds'). Below the form are four buttons: a green '+' icon labeled 'Add', a red minus icon labeled 'Delete', a blue circular arrow icon labeled 'Reset', and a blue checkmark icon labeled 'Save'.

productid	description	price	productname
1	nice mouse	19	wireless mouse
2	slick keyboard	39	ergonomic keyboard
3	best screen saver	15.100000381469727	screen saver
4	100 cds	22	case of cds
5	flash memory	5	memory stick

The client application uses the deployed data model to bind data to its controls. In the previous illustration, notice that data retrieved from the Product table is displayed in a DataGrid control:

Complete the following steps to create a Flex client:

- 1 Start the samples database if it isn't already running.
- 2 Start your application server if it isn't already running.
- 3 Start Flash Builder if it isn't already running.
- 4 In Flash Builder, switch to Design view in the Flash perspective.
- 5 In the Package Explorer, expand the nodes of your Flex project to src > (default package) and open the displayed MXML file.
- 6 Select the Components tab in the lower left side of the Flash Builder window.
- 7 Expand the Controls folder in the Components tree.
- 8 Drag the DataGrid control to the Design view canvas.
- 9 Set the editable property of the DataGrid to true.
- 10 Set the width of the DataGrid to 100%.
- 11 Select the Data/Services tab under the Design view canvas.
- 12 In the Data/Services view, expand the ProductService node.
- 13 Drag the `getAll() : Product []` function to the DataGrid control. Accept the default Bind To Data settings. The DataGrid columns changes to match the property names of the Product entity.
- 14 Right click the DataGrid control and select Generate Details Form.
- 15 Select Model Driven Form and click OK to generate a Form.
- 16 Reposition the form so that it is not covering the DataGrid control.
- 17 Run the application. The DataGrid control should fill with data; the `productService.getAll()` function is called in the DataGrid control's creationComplete event handler.

Note: Calling the `getAll()` function results in a call to the `fill()` method of the underlying DataService instance.
- 18 Select a row in the DataGrid control to display details in the form.
- 19 Click Delete to delete an item from the database.

Note: Deleting an item with the form results in a call to the `deleteItem()` method of the underlying `DataService` instance.

When you delete an item, the Java code located in the assembler's `deleteItem()` method tracks the date and time that the item was deleted. If you delete row 4, a log file is created that contains the following information:

```
The case of cds items priced at 22.0 was deleted at: 2010-02-17 17:40:08.
```

Implementing an entity method on the server

You can add custom functionality to the server by adding new methods to the generated entity Java file. For example, assume that you create business logic that discounts the price of items located in the Product table. You can add a new property to the data model that stores the discounted price. To perform this task, modify the Product data model file by adding a property element and method element to it, as shown in the following example:

```
<property name="discountPrice" type="float" expr="discount(price)">
</property>
<method return-type="float" arguments="p:float" name="discount"></method>
```

Note: You can add these elements right before the closing `</entity>` tag. This adds a property named `discountPrice` to the `Product` entity. Before reading this section, it is recommended that you first perform the steps specified in “[Implementing an assembler method on the server](#)” on page 350. To reference this property on in the client application, you need to implement the `discount` function in the ActionScript generated class. For information about modifying client application logic, see “[Customizing client-side functionality](#)” on page 341.

Notice that the property element specifies a derived property named `discountPrice`. The derived property is populated by using the `expr` attribute. In this example, the value of the `expr` attribute is `discount(price)`. This value references a method defined in the method element. When you generate Java source files by using the data model that contains these elements, the super class contains a method named `discount()`.

```
protected Float discount(Float p)
{
    throw new UnsupportedOperationException("Function discount must be overridden");
}
```

You write the implementation of this method in the entity class, which extends the super class. That is, when the `_Super_Product` class is generated, it contains the `discount()` method. You override the `discount()` method in the `Product` class (the entity class) by defining the application logic.

The following example shows the implementation of the `discount()` method located in the `Product` class.

```
import javax.persistence.*;
@Entity
@Table(name="PRODUCT")
public class Product extends _Super_Product
{
    public Float discount(Float p)
    {
        //discount the price by 5
        Float myPrice = p - 5;
        return myPrice ;
    }
}
```

The return value of the `discount()` method represents the discount price.

Note: To keep this example straightforward, the `discount()` method returns a static value by reducing the price by the value five. However, you can add application logic to dynamically generate a value. For example, you can add application logic that retrieves a value from a relational database.

You can retrieve the discount price value by creating an instance of Product. From within the Assembler class, you can cast `item` to Product. Then you can invoke the Product instance's `discountPrice()` method to get the discount price value. The following code example shows the modified `deleteItem()` method in the `ProductAssembler` class. The lines of code in bold represent the application logic that retrieves the discount price value.

```
public void deleteItem(Object item)
{
    try
    {

        //Get the description and price of the item being deleted
        Product myProduct = (Product)item;
        Float myPrice = myProduct.getPrice();
        String strPrice = myPrice.toString();
        String strName = myProduct.getProductName();

        File myFile = new File("C:\\log.txt");
        BufferedWriter out = new BufferedWriter(new FileWriter(myFile));
        System.out.println(strName);

        //Do not allow flash memory to be deleted
        if (strName.equals("memory stick"))
        {
            //Write a message to the log file that flash memory cannot be deleted
            out.write("A record where item is memory stick cannot be deleted");
            out.close();

            //Throw an Exception
            Exception myException = new Exception("A record where item is memory
                stick cannot be deleted.");
            throw myException;
        }

        //Get the time when the item is deleted
        String DATE_FORMAT_NOW = "yyyy-MM-dd HH:mm:ss";
        Calendar cal = Calendar.getInstance();
        SimpleDateFormat sdf = new SimpleDateFormat(DATE_FORMAT_NOW);
        String myTime = sdf.format(cal.getTime());
        //Get the value of the discount price
        float discountVal = myProduct.discount(myPrice);
        //Write out the values to the log file
        out.write("The "+strName+ " item priced at "+ strPrice +" was deleted at:
        "+myTime);
        out.close();
        super.deleteItem(item);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

In the previous code example, the value of discount price was retrieved by invoking the Product object's `discountPrice()` method. Instead of calling the method directly, you can use an EntityUtility object as shown in the following code example. The EntityUtility is available from the superclass, FiberAssembler.

```
EntityUtility eu = entityUtilityFactory.getEntityUtility(entity);
Float discountPrice = (Float) eu.getPropertyValue("discountPrice", item);
```

Both ways return the same value. To implement an entity method on the server, follow the same steps that are specified in “[Implementing an assembler method on the server](#)” on page 350. However, there are a few differences. First after your create data model, ensure that you add the property element and method element to it as discussed earlier.

Next, when you modify the Java source files, modify both the Product class and the ProductAssembler class. In the Product class, create the application logic for the `discountPrice()` method. In the ProductAssembler class, create an instance of Product and invoke the `discountPrice()` method.

After you modify the Java source files, modify the data model and redeploy it to the server. (See “[Modify the data model](#)” on page 356.)

Add the following DMS annotation to your data model to ensure that the modified Product class is used:

```
<item name="ServerGeneratedEntites">false</item>
```

Also ensure that the following annotation is present on the Product entity so that your ProductAssembler class is used:

```
<annotation name="DMS">
<item name="AssemblerClass">Product.ProductAssembler</item>
</annotation>
```

The following code example shows the FML file that contains the new annotations. The new lines are in bold.

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <annotation name="DMS">
        <item name="datasource">java:/comp/env/jdbc/ordersDB</item>
        <item name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</item>
        <item name="ServerGeneratedEntities">false</item>
    </annotation>
    <annotation name="group"/>
    <entity name="Product" persistent="true">
        <annotation name="ServerProperties">
            <item name="ServerType">LCDS</item>
        </annotation>
        <annotation name="DMS">
            <item name="AssemblerClass">Product.ProductAssembler</item>
            <item name="Table">PRODUCT</item>
        </annotation>
        <annotation name="VisualModeler">
            <item name="width">115</item>
            <item name="height">110</item>
            <item name="x">10</item>
            <item name="y">10</item>
        </annotation>
        <id name="productid" type="integer">
            <annotation name="DMS">
                <item name="ColumnName">PRODUCTID</item>
            </annotation>
        </id>
    </entity>
</model>
```

```
<property name="description" type="string" length="255">
    <annotation name="DMS">
        <item name="ColumnName">DESCRIPTION</item>
    </annotation>
</property>
<property name="price" type="float">
    <annotation name="DMS">
        <item name="ColumnName">PRICE</item>
    </annotation>
</property>
<property name="productname" type="string" length="255">
    <annotation name="DMS">
        <item name="ColumnName">PRODUCTNAME</item>
    </annotation>
</property>
<property name="discountPrice" type="float" expr="discount(price)">
</property>
<method return-type="float" arguments="p:float" name="discount"></method>
</entity>
</model>
```

You can use the Modeler in Flash Builder to add these annotations to the data model. For more information about using the Modeler in Flash Builder, see the *see the Application Modeling Technology Reference*.

Create the client application using Flash Builder. When you delete an item, the value of the discount price property is written to the TXT file. (See “[Create and run a Flex client](#)” on page 357.)

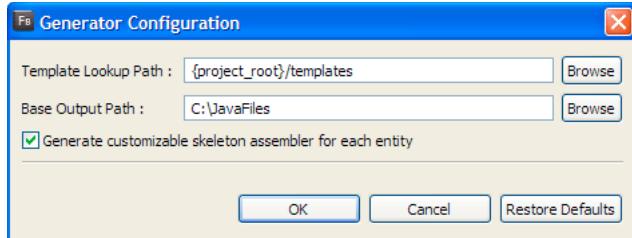
Modifying Java template files

You can modify template files that are used to generate the Java source files. After you modify the template files and generate the Java source files, the modifications appear in the generated Java source files. The template files are FreeMarker files; for more information, see [freemarker.org](#).

Each Java source file has a corresponding template file. You can extract and customize the following template files:

Template file	Description
JavaAssemblerSkeleton.ftl	Generates <i>EntityNameAssembler.java</i>
JavaBeanInfo.ftl	Generates <i>EntityNameBeanInfo.java</i>
JavaEntityBeanChild.ftl	Generates <i>EntityName.java</i>
JavaEntityBeanSuper.ftl	Generates <i>_Super_EntityName.java</i>
JavaldBean.ftl	Generates <i>EntityNamePK.java</i>

The template files are searched for in the location specified in the Template Lookup Path field (see the following illustration). If they are not found in this location, then default values are used. The lookup directory is where the extract operation places the files. The following illustration shows the Template Lookup Path field.



In this example, the template files are written to the C:\JavaFiles\templates directory.

Note: You are prompted to replace an existing template file before it is overwritten.

(Optional) To change the template lookup path:

- 1 From the Flash Builder menu, select Window > Preferences.
- 2 In the Preferences dialog, select Adobe > Data Model > Code Generation.
- 3 Click Configure Generator for Server-side Java Generation.
- 4 Enter a valid path in the Template Lookup Path field.
- 5 Click OK.

The templates use the following tokens that correspond to Java classes to perform code generation. See the LiveCycle Data Services Javadoc documentation for more information about the corresponding Java APIs.

Template token	Description
model	Corresponds to fiber.core.api.Model
entity	Corresponds to fiber.core.api.Entity
util	Corresponds to fiber.gen.java.utils.JavaGenerationUtil
packageMap	Corresponds to Map of String to String: entity names to package names
typeHelper	Corresponds to fiber.gen.java.wrapper.JavaTypeHelper
termFormatter	Corresponds to fiber.gen.java.JavaTermFormatter

To better understand the templates, see the related Javadoc and examine the source code of the generated classes.

Modify Java template files:

- 1 From the Flash Builder menu, select Window > Preferences.
- 2 In the Preferences dialog, select Adobe > Data Model > Code Generation.
- 3 Select Default from the Java drop-down list. (By default, this value is None.)
- 4 Click Configure Generator.
- 5 Specify the directory location in which template files are written in the Template Lookup Path field.
- 6 Click OK.

- 7 In Project Explorer, right click your data model file. From the pop-up menu, select Data Model > Extract Templates For Active Generators.
- 8 Open a template file from the location specified in the Template Lookup Path field.
- 9 Modify the FreeMarker template file to meet your business requirements.
- 10 Generate Java source files. (See “[Generate Java source files based on the data model](#)” on page 353.)

Note: There is no guarantee that your customized template and generated code will work in future releases of LiveCycle Data Services. Also, to apply a LiveCycle Data Services patch or update, you must extract the templates again.

An example of modifying the template files

Assume, for example, that you want to modify the JavaAssemblerSkeleton.ftl so that only the `deleteItem()` method appears. In this situation, you can open the JavaAssemblerSkeleton.ftl file in a text editor and remove all methods except for the `deleteItem()` method, as shown in the following example.

```
<!-- ****
*
* ADOBE CONFIDENTIAL
*
*
* Copyright 2010 Adobe Systems Incorporated
* All Rights Reserved.
*
* NOTICE: All information contained herein is, and remains
* the property of Adobe Systems Incorporated and its suppliers,
* if any. The intellectual and technical concepts contained
* herein are proprietary to Adobe Systems Incorporated and its
* suppliers and may be covered by U.S. and Foreign Patents,
* patents in process, and are protected by trade secret or copyright law.
* Dissemination of this information or reproduction of this material
* is strictly forbidden unless prior written permission is obtained
* from Adobe Systems Incorporated.
* -->
<if packageMap.get(entity.getName() + "__Assembler")??>
    <#assign packageName = packageMap.get(entity.getName() + "__Assembler") />
<#elseif packageMap.get(entity.getName())??>
    <#assign packageName = packageMap.get(entity.getName()) />
<#else>
    <#assign packageName = model.name />
</if>
<#assign type=entity.getName() />
<#assign classname=util.formatAssemblerName(type) />
<!-- Body of generated value object -->
<#assign classbody>
public class ${classname} extends
${typeHelper.addImport("fiber.data.assemblers.FiberAssembler", false)}
{

    /**
     * Deletes an item.
     *
     * @param item the item to delete.
     */
    @Override
    public void deleteItem(Object item)
{
```

```
        super.deleteItem(item);
    }
}
</#assign>
<!-- ===== -->
->
<!-- Package declaration and imports -->
/** 
 * Skeleton Model Driven Assembler which you can use to customize behavior.
 * This assembler manages the ${entity.getName()} entity.
 *
 * You do not need to use this class unless you need to customize the server
 * side logic of an assembler operation (e.g. any of the CRUD operations).
 *
 * This class is only generated when there is no file already present
 * at its target location. Thus custom behavior that you add here will
 * survive regeneration if the model is modified.
 *
 * The following annotation must be present on the entity in the model
 * in order for this class to be used by the server:
 *
*   <entity name="${entity.getName()}">
*     <annotation name="DMS">
*       <item name="AssemblerClass">${packageName}.${classname}</item>
*     </annotation>
*
* If this annotation is not present, the server will use the default
* model driven assembler, even if this class is available on the server.
*
* Before deploying a model with the AssemblerClass value set,
* you will need to compile this class (and any associated classes) and place
* them in the classpath of the LCDS web application.
*
*/
package ${packageName};
<#list typeHelper.getImports() as imp>
    <if imp??>
import ${imp};
    </if>
</list>
import fiber.data.services.ModelValidationException;
import java.util.List;
import java.util.Map;
import java.util.Collection;
import flex.data.PropertySpecifier;
${classbody}
```

After you modify the template files and generate the Java source files, the modifications appear in the generated Java source files. For example, the ProductAssembler class contains only the deleteItem() method.

Generating database tables from a model

You can create a model from which you can generate tables in a SQL database and Model Assembler code for working with the database from a Flex client.

Complete these steps to generate database tables from a model:

- 1 Create a database in your database server or choose an existing database to work with.
- 2 Copy the driver JAR file for your database server to the WEB-INF/lib directory of your web application (for example, the lcds-samples web application).
- 3 In the Modeler, create a model in your Flex project by clicking the Open Model for Active Project icon at the right end of the Package Explorer icon bar. The new model is displayed.
- 4 Add the following annotation below the `<model ...>` element. The datasource value must match the name of the JDBC data source you defined on your application server.

```
<annotation name="DMS">
    <item name="datasource">java:/comp/env/jdbc/your_database_name</item>
</annotation>
```

- 5 Below the annotation, add entity elements with the ServerProperties annotation shown in the following example. Application modeling technology generates a corresponding database table and Model Assembler destination for each entity. The Table annotation is optional; if not specified, the generated table name matches the entity name.

```
<entity name="Thing1" persistent="true">
    <annotation name="ServerProperties">
        <item name="ServerType">LCDS</item>
    </annotation>
    <annotation name="DMS">
        <item name="Table">THING1</item>
    </annotation>
</entity>
```

- 6 Add at least one id element to each entity and add zero or more property elements to the entities. Application modeling technology generates database columns for each id and property in an entity. The id and property annotations are optional; if not specified, the generated table column names match the id and property names.

```
<entity name="Thing1" persistent="true">
    <annotation name="ServerProperties">
        <item name="ServerType">LCDS</item>
    </annotation>
    <annotation name="DMS">
        <item name="Table">THING1</item>
    </annotation>
    <id name="orderlineid" type="integer">
        <annotation name="DMS">
            <item name="ColumnName">THING1ID</item>
        </annotation>
    </id>
    <property name="quantity" type="integer">
        <annotation name="DMS">
            <item name="ColumnName">QUANTITY</item>
        </annotation>
    </property>
</entity>
```

Within persistent entities, an id property is a data property, the value of which represents part or all the identity of the entity instance. Persistent entities must specify at least one id property. A property element can contain a data property of any valid type, or a derived property for which the value is computed based on the values of data properties.

- 7 Save the model to generate ActionScript service wrappers and value objects that represent each entity. You can see these objects in the Flash Builder Data/Services panel.

Note: There are several ways to create channels for client applications to contact the server. As a best practice, manually create a channel set in your client MXML application, as described in “[Configuring channels and endpoints](#)” on page 39. Alternatively, you can add channel set annotations in the model; for more information, see “Annotations for client-side generation” in [Application Modeling Technology Reference](#). A default channel is used if one is configured on the server and you do not create a channel set in MXML or the model.

- 8 Click the Deploy Model to LCDS Server icon on the Modeler icon bar to deploy the model to the LiveCycle Data Services server.
- 9 To the right of Database Tables, select Recreate. Click Finish. The model is deployed to the server and database tables and Model Assembler destinations are generated on the server.

You can now populate the generated database tables using one of these methods:

- From a Flex client application that you create in Flash Builder
- Manually
- With a database script

Complete the following steps to quickly build a simple Flex user interface to populate your database and then display data in the database.

Important: The model-driven form, which is used in this procedure, is a developer productivity aid only. It is not a general solution for RIA form generation and development. Regenerating a model-driven form overwrites the existing form.

- 1 Open the default MXML file of your Flex project into the Flash Builder Design view.
- 2 From the Components panel, drag a DataGrid control onto the Design view stage.
- 3 In the Data/Services panel, drag the `getAll()` operation onto the DataGrid control. Dragging the operation to the DataGrid control automatically binds the operation result to the DataGrid control.
- 4 In the Data/Services panel, right-click one of the listed data types and select Generate form > Model Driven Form.
- 5 Go to the Source view and find the code that references the generated form at the bottom of the MXML file. The form is an MXML component saved in the `src/ProjectName/forms` directory of the Flex project.
- 6 Change the value of the form's `valueObject` attribute to: `valueObject="{dataGrid.selectedItem as YourDataType}"` where `YourDataType` is the name of the data type you are working with.
- 7 Save the MXML application and run it.
- 8 You can use the Add button to add a new item of your specified data type. When you click the Save button, a new instance of the data type is created on the client. On the server, a new instance of the data type is created and the database is updated accordingly.
- 9 After adding and saving some items, refresh the browser to see that the items now exist in the database and are sent back to the client.
- 10 Click an item in the DataGrid control and the form displays the details for that item.

Creating a client for an existing service destination

You can create a model-driven client that uses existing service destinations on the LiveCycle Data Services server. For this type of application you use a Flash Builder service wizard to generate a model and corresponding ActionScript code. This type of development is useful when you have existing destinations.

Unlike entities that you use with model-driven destinations, you cannot deploy entities for existing destinations to the server, so you cannot apply server-side (DMS) annotations. You can apply ActionScript-generation annotations that apply only to the client. Note however that a model that contains entities generated from an existing destination is overwritten if you re-import the destination; all manual changes are lost.

Note: *You can combine entities that you create for model-driven destinations and entities for existing (not model-driven) destinations in the same model. However, you cannot create relationships between the two types of entities.*

Summary of steps

Perform the following tasks to build a model-driven client for an existing service destination:

- 1 Install [LiveCycle Data Services](#) and [Flash Builder](#).
- 2 “[Configure RDS on the server](#)” on page 328 (Common with the *Build your first model-driven application* section.)
- 3 “[Configure Flash Builder to use RDS](#)” on page 329 (Common with the *Build your first model-driven application* section.)
- 4 “[Create a Flex project in Flash Builder](#)” on page 329 (Common with the *Build your first model-driven application* section.)
- 5 Import an existing service destination with a service wizard; you can import remoting, data management, and web service destinations. (Specific to this workflow.)
- 6 “[Building the client application](#)” on page 372 (Common to general development.)

Import an existing remoting or data management destination

Complete these steps to generate ActionScript code for an existing Remoting Service or Data Management Service destination on the server. The steps are slightly different for web service destinations; for more information, see “[Importing an existing web service destination](#)” on page 368.

To import an existing Data Management Service destination configured in the data-management-config.xml file, the destination must specify an `item-class` element. At least one `fill-method` element is required when the assembler does not implement the Assembler interface. You cannot use this feature with destinations that use the SQL Assembler. The following example shows a destination for an assembler that implements the Assembler interface.

```
<destination id="crm-company">
    <properties>
        <source>flex.samples.crm.company.CompanyAssembler</source>
        <scope>application</scope>
        <item-class>flex.samples.crm.company.Company</item-class>
        <metadata>
            <identity property="companyId" />
        </metadata>
    </properties>
</destination>
```

- 1 From the Flash Builder Data menu, select Connect To Data/Service.

Flash Builder provides various ways to connect to the data service:

- Flash Builder Data menu
- Data/Services view Data menu
- Data/Services view context menu

- 2 In the Select Service Type dialog, select LCDS. Click Next.

- 3 If you are prompted for a username and password, select No Password Required.
- 4 In the Import BlazeDS/LCDS Service dialog, select one of the listed destinations. For demonstration purposes, select the ProductService Remoting Service destination.
- 5 Click Finish.

Flash Builder generates ActionScript classes that provide access to service operations from the client application. The operations for the service are available in the Data/Services view.

Note: When connecting to a data service, Flash Builder needs to know the data type for the data returned by a service operation. Many services define the type of returned data on the server (server-side typing). However, if the server does not define the type, then the client application must configure the type for returned data (client-side typing). Flash Builder indicates if this is required when you attempt to bind an operation to a user interface component. For more information, see “[Configuring return types for a data service operation](#)” on page 376.

- 6 See “[Building the client application](#)” on page 372 for information on building a Flex client that uses the ActionScript code generated from the model.

Note: You can enable client-side data management features for Flex applications that access Remoting Service destinations. Client-side data management allows the synchronization of data updates on the server from the client application. You can modify one or more items in a client application without making any updates to the server. You then commit all the changes to the server with one operation. You can also revert the modifications without updating any data.

Importing an existing web service destination

- 1 From the Flash Builder Data menu, select Connect To Data/Service.

Flash Builder provides various ways to connect to the data service:
 - Flash Builder Data menu
 - Data/Services view Data menu
 - Data/Services view context menu
- 2 In the Select Service Type dialog, select WebService. Click Next.
- 3 In the Specify WSDL to Introspect dialog, enter a service name.
- 4 Select a destination from the Destination list.
- 5 Under How Will Your Application Access The Web Service?, select Through A LiveCycle Data Service Proxy Destination.
- 6 Click Next.
- 7 Select the web service operations you want to import. Click Finish.

Flash Builder generates ActionScript classes that provide access to service operations from the client application. The operations for the service are available in the Data/Services view.
- 8 See “[Building the client application](#)” on page 372 for information on building a Flex client that uses the ActionScript code generated from the model.

Manage data access from the server

Note: This content does not apply the Flex applications that use the Data Management Service in LiveCycle Data Services. The Data Management Service provides server-side data management. This content describes client-side data management and paging for RPC services.

Paging Paging is the incremental retrieval of large data sets from a remote service.

For example, suppose you want to access a database that has 10,000 records and then display the data in a DataGrid control that has 20 rows. You can implement a paging operation to fetch the rows in 20 set increments. When the user requests additional data (scrolling in the DataGrid control), the next page of records is fetched and displayed.

Data management In Flash Builder, data management is the synchronization of updates to data on the server from the client application. Using data management, you can modify one or more items in a client application without making any updates to the server. You then commit all the changes to the server with one operation. You can also revert the modifications without updating any data.

Data management involves coordinating several operations (create, get, update, delete) to respond to events from the client application, such as updating an Employee record.

Enabling paging

To enable paging your data service must implement a function with the following signature:

```
getItems_paged(startIndex:Number, numItems:Number) : myDataType
```

function name	You can use any valid name for the function.
startIndex	The initial row of data to retrieve. The data type for startIndex should be defined as Number in the client operation.
numItems	The number of rows of data to retrieve in each page. The data type for numItems should be defined as Number in the client operation.
myDataType	The data type returned by the data service.

When implementing paging from a service, you can also implement a `count()` operation. A `count()` operation returns the number of items returned from the service. The `count()` operation must implement the following signature:

```
count() : Number
```

function name	You can use any valid name for the function.
Number	The number of records retrieved from the operation.

Flex uses the `count()` operation to properly display user interface components that retrieve large data sets. For example, the `count()` operation helps determine the thumb size for a scroll bar of a DataGrid control.

Some remote services do not provide a `count()` operation. Paging still works without a count operation, but the control displaying the paged data might not properly represent the size of the data set.

Enable paging for an operation

This procedure assumes that you have coded both `getItems_paged()` and `count()` operations in your remote service. It also assumes that you have configured the return data type for the operation, as explained in “[Configuring return types for a data service operation](#)” on page 376.

- 1 In the Data/Services view, from the context menu for the `getItems_paged()` operation, select Enable Paging.
- 2 If you have not previously identified a unique key for your data type, specify the attributes that uniquely identify an instance of this data type. Click Next.

Typically, this attribute is the primary key.

- 3 Specify the `count()` operation. Click Finish.

Paging is now enabled for that operation.

In Data/Services view, the signature of the function that implements paging no longer includes the `startIndex` and `numItems` parameters. These values are now dynamically added based on the user interface component that displays the paged data.

Enabling client-side data management

Note: This content does not apply the Flex applications that use the Data Management Service in LiveCycle Data Services. The Data Management Service provides server-side data management. This content describes client-side data management.

To enable data management, implement one or more of the following functions, which are used to synchronize updates to data on the remote server:

- Add (`createItem`)
- Get All Properties (`getItem`)
- Update (`updateItem`)
- Delete (`deleteItem`)

These functions must have the following signatures:

```
createItem(item:myDatatype) : int
deleteItem(itemID:Number) : void
updateItem((item: myDatatype) : void
getItem(itemID:Number) : myDatatype
```

function name	You can use any valid name for the function.
item	An item of the data type returned by the data service.
itemID	A unique identifier for the item, usually the primary key in the database.
myDataType	The data type of the item available from the data service. Typically, you define a custom data type when retrieving data from a service.

Enable data management for an operation

This procedure assumes that you have implemented the required operations in your remote service. It also assumes that you have configured the return data type for the operations that use a custom data type. See “[Configuring return types for a data service operation](#)” on page 376.

- 1 In the Data/Services view, expand the Data Types node.
- 2 From the context menu for a data type, select Enable Data Management.
- 3 If you have not previously identified a unique key for your data type, specify the attributes that uniquely identify an instance of this data type. Click Next.

Typically, this attribute is the primary key.
- 4 Specify the Add, Get All Properties, Update, and Delete operations. Click Finish.

Data management is now enabled for that operation.

Configuring a data source

End-to-end model-driven development lets you generate Data Management Service destinations from models. To use this functionality, configure a JDBC data source for each SQL database that you want to use. The RDS server uses this data source to display database tables in the RDS Dataview panel of the Modeler. For more information about the RDS server, see “[Configure RDS on the server](#)” on page 328.

How you configure a JDBC data source depends on the application server you use. For more information, see your application server documentation.

On the Apache Tomcat server, you configure JDBC data sources in a context file. The default context file is tomcat/conf/context.xml. You can also configure data sources in a context file for a specific host and web application; for example, tomcat/conf/Catalina/localhost/web_app_name.xml file.

The following example shows a JDBC data source configuration:

```
<Context privileged="true" antiResourceLocking="false"
    antiJARLocking="false" reloadable="true">
    <Resource name="jdbc/ordersDB" type="javax.sql.DataSource"
        driverClassName="org.hsqldb.jdbcDriver"
        maxIdle="2" maxWait="5000"
        url="jdbc:hsqldb:hsq1://localhost:9002/ordersdb"
        username="sa" password="" maxActive="4"/>
</Context>
```

Configuring RDS on the server

The lcds and lcds-samples web applications in LiveCycle Data Services include an RDS servlet that is preconfigured but commented out in the WEB-INF/web.xml file.

For local development, you can use the RDS servlet without configuring application server security. However, this is not recommended for any other type of deployment.

The following example shows an RDS servlet configuration in the WEB-INF/web.xml file with an init-param setting that bypasses application server security:

```
<servlet>
    <servlet-name>RDSDispatchServlet</servlet-name>
    <display-name>RDSDispatchServlet</display-name>
    <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-class>
        <init-param>
            <param-name>useAppserverSecurity</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>10</load-on-startup>
    </servlet>
    <servlet-mapping id="RDS_DISPATCH_MAPPING">
        <servlet-name>RDSDispatchServlet</servlet-name>
        <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
    </servlet-mapping>
```

By default, the RDS server uses application server security, and you configure a role named rds and assign a user to that role.

The following example shows the default RDS servlet configuration in the WEB-INF/web.xml file. Uncomment this part of the file.

```
<servlet>
    <servlet-name>RDSDispatchServlet</servlet-name>
    <display-name>RDSDispatchServlet</display-name>
    <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-class>
    <load-on-startup>10</load-on-startup>
</servlet>
<servlet-mapping id="RDS_DISPATCH_MAPPING">
    <servlet-name>RDSDispatchServlet</servlet-name>
    <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
</servlet-mapping>
```

How you configure application server security depends on the application server you use. For more information, see your application server documentation.

On the Apache Tomcat server, by default you configure users and roles in the tomcat-users.xml file in the tomcat/conf directory. The following example shows a user and role configured for RDS in a tomcat-users.xml file:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
    <role rolename="rds"/>
    <user username="rdsuser" password="rdspassword" roles="rds"/>
</tomcat-users>
```

You must also configure custom authentication. For example, on Apache Tomcat, complete the following steps:

- 1 Locate the Tomcat security resource libraries under install_root/resources/security/tomcat. Place the flex-tomcat-common.jar and flex-tomcat-server.jar files in the tomcat/lib directory.
- 2 Create or edit a context file for your web application. For example, for the lcds-samples web application, create or add to an existing tomcat-root/conf/Catalina/localhost/lcds-samples.xml file. In that file, add the following line:

```
<valve classname="flex.messaging.security.TomcatValve"></valve>
```
- 3 For non-turnkey installations, make sure the TomcatLoginCommand is active in the <SECURITY> section of the services-config.xml file:

```
<security><login-command server="Tomcat"></login-command>
...
</security>
```

Building the client application

Use the Flash Builder code editor to create a user interface. You can use the editor either in Design view or Source view. You can also generate a form from service operations in the Data/Services view. You can generate a simple Flex form or a model-driven form that takes advantage of advanced application modeling technology features provided on the LiveCycle Data Service server.

Important: *The model-driven form is a developer productivity aid only. It is not a general solution for RIA form generation and development. Regenerating a model-driven form overwrites the existing form.*

After laying out the components for the application, generate event handlers as needed for user interaction with the application.

Using Flash Builder Design view to build an application

In Design view of the MXML editor drag-and-drop components from the Components view into the Design Area. Then arrange the components and configure their properties. After designing the application layout, bind data returned from the data service to the components.

Controls

Flex provides a rich set of user-interface components such as Button, TextArea, and ComboBox controls. You place controls in containers. Containers are user-interface components that provide a hierarchical structure for controls and other containers.

Flex provides various data-driven controls that are ideal for displaying sets of data in lists, tables, or trees.

- [DataGrid](#)
- [AdvancedDataGrid](#)
- [OLAPDataGrid](#)
- [List](#)
- [TileList](#)
- [Tree](#)

Binding service operations to controls

There are several ways to bind a service operation to a control component. You can drag service operation from the Data/Service view onto a component in Design view. You can also open the Bind to Data dialog to select an operation to bind to a component.

When you bind a service operation to a control component, Flash Builder generates MXML and ActionScript code to access the service operation from the client application.

Bind a service operation to a DataGrid control (drag-and-drop)

This procedure assumes that you have connected to a data service. It also assumes that you have configured the data return type for a service operation so it returns a set of items.

- 1 In Design view, drag a DataGrid control from Components view to the editing area.
 - 2 Drag an operation from the Data/Services view onto the DataGrid control.
The DataGrid control changes to show the fields retrieved from the database.
 - 3 Customize the display of the DataGrid control.
 - 4 Save and run the application.

Bind a DataGrid control to a service operation (Bind to Data dialog)

This procedure assumes that you have connected to a data service and the data return types for operations are configured.

- 1 In Design view, drag a DataGrid control from Component view to the editing area.
 - 2 Open the Bind to Data dialog using one of the following methods:
 - Select Bind to Data from either the Flash Builder Data menu or the DataGrid context menu.

- Select the Bind to Data button in the Property Inspector.

3 Select New Service Call, then select a Service and Operation.

4 (Optional) Select Change Return Type:

Select Change Return Type if you want to reconfigure the return type for the service operation.

If the return type for the operation has not configured appropriately, select Configure Return Type to continue.

5 Click OK.

The DataGrid control changes to show the fields retrieved from the service.

6 Customize the display of the DataGrid control.

7 Save and run the application.

Generating a form for an application

Flash Builder can generate several types of forms for accessing data services.

Form	Description
Custom data type	Contains entries for each field of a custom data type for a service
Master -detail form	Contains entries for a selected item in a data control, such as a DataGrid or a List
Service call	Displays the data returned from a service call

When generating forms, specify which fields to include and whether to make the form editable. You can also specify the type of user interface control to represent the form items.

When working with end-to-end model-driven development, you can generate a model-driven form that takes advantage of advanced application modeling technology features provided on the LiveCycle Data Service server.

Important: *The model-driven form is a developer productivity aid only. It is not a general solution for RIA form generation and development. Regenerating a model-driven form overwrites the existing form.*

Generating a form for a service call

This procedure assumes that you have connected to a data service and that you are in Design view of the MXML Editor.

This procedure shows how to generate a form for a service call. The procedure for generating other types of forms are similar.

1 There are several ways to run the Generate Form wizard:

- From the Data/Service view, select an operation from the context menu. Select Generate Form.
- From the Flash Builder Data menu, select Generate Form or Generate Details Form.
- Select a Data Control in Design view. From the context menu, select Generate Details Form.

2 In the Generate Form wizard, select Generate Form For Service Call.

3 Select New Service Call, then specify the Service and Service Operation.

4 Configure the return type (or change the return type) for the selected operation if necessary.

The return type for the operation must be configured before you can generate the form. If you previously configured a return type, you have the option to change the return type.

See “[Configuring return types for a data service operation](#)” on page 376.

- 5 Specify whether to include forms for input parameters and return type.
- 6 Specify whether to make the form editable. Click Next.
- 7 In the Property control Mapping dialog, select which fields to include in the form and the type of control to represent the data.
- 8 Click Finish.

Generating a model-driven form

Important: The model-driven form is a developer productivity aid only. It is not a general solution for RIA form generation and development. Regenerating a model-driven form overwrites the existing form.

Follow this procedure to generate a model-driven form that takes advantage of advanced application modeling technology features provided on the LiveCycle Data Services server.

A model-driven form that has a *-to-one association to a non-model-driven assembler generates a line of code similar to the following, in which you must manually enter valid fill parameters:

```
var accountCategoryToken:AsyncToken = new  
    AccountCategoryService() .fill(INSERT_CORRECT_FILL_PARAMETERS_HERE);
```

If you try to compile the application without entering fill parameters, the compiler issues the following error:

```
Access of undefined property INSERT_CORRECT_FILL_PARAMETERS_HERE
```

This procedure assumes that you have connected to a data service and that you are in Design view of the MXML Editor. This procedure shows how to generate a form for a data type. The procedure for generating other types of forms are similar.

- 1 Right-click a DataGrid control to which you have bound data and select Generate Details Form.
- 2 Select Model Driven Form and click OK to generate a Form.
- 3 Reposition the form so that it is not covering the DataGrid control.
- 4 Run the application. The DataGrid control should fill with data; the `productService.getAll()` function is called in the DataGrid control’s creationComplete event handler.

Note: Calling the `getAll()` function results in a call to the `fill()` method of the underlying `DataService` instance.

- 5 Select a row in the DataGrid control to display details in the form.

Using global styles with a model-driven form

If you generate a model-driven form for an entity with a property that uses a global style, at run time the form label for that property is replaced with the style’s caption text. The default caption message is “Caption.” To work around this issue, you can use one of three options:

- Delete the caption message of the global style in Source view
- Use an inline style instead of a global style
- Use a bundle and key instead of the caption text property

Using required properties with a model-driven form

When a model-driven form is generated from an entity that contains a required property, the generated ActionScript code does not check for empty strings. To ensure that a property is required and that empty data is not sent to the server, in the model set required attribute of the property to true and also use a constraint element that constrains the length of the property to a minimum length.

Generating event handlers

When you bind a data service operation to a component, Flash Builder generates an event handler that retrieves data from the service to populate the component.

For example, if you bind an operation such as `getAllItems()` to a DataGrid control, Flash Builder generates a `creationComplete` event handler.

```
<DataGrid creationComplete="getAllItemsResult.token = productService.getAllItems() " ... >
```

When you run the application, once the DataGrid control has been created, it populates itself with data retrieved from the service.

You can accept the generated event handlers or replace them with others according to your needs. For example, you can replace the `creationComplete` event handler on the DataGrid control with a `creationComplete` handler on the Application.

You can also generate or create event handlers for controls that accept user input, such as Buttons or Text. Do either of the following to generate event handlers:

- From the Data/Services view, drag an operation onto the control.

Flash Builder generates an event handler for the default event for the component. For example, for a Button, the event handler would be the Click event.

- In Design view, select the control and then in the Property Inspector, click the generate event icon.

Flash Builder opens the Source view of the editor and generates a stub for the event handler.

Fill in the remaining code for the event handler. Flash Builder provides Content Assist to help you code the event handler.

Configuring return types for a data service operation

When connecting to a data service, Flash Builder needs to know the data type for the data returned by a service operation. The data types supported are those types recognized by AMF.

Many services define the type of returned data on the server (server-side typing). This is always the case when working Data Management Service destinations. However, if the server does not define the type, then the client application must configure the type for returned data (client-side typing).

When configuring return types for client-side typing, the data type can be an ActionScript data type, a custom data type representing complex data, or void to indicate the operation does not return any data. The following table lists possible data types for client-side typing.

Data Type	Description
ActionScript types	Boolean Boolean[] Date Date[] int int[] Number Number[] String String[]
No data returned	void
User-defined type	<i>CustomType</i> <i>CustomType[]</i>

Typically, you define custom data types for service operations that return complex data.

Flash Builder code generation

Flash Builder generates client code that provides access to remote service operations. Flash builder generates code in the following circumstances:

- Saving a model in a Flash Builder project.
- Refreshing the data service in Data/Services view
- Configuring a return type for an operation
- Binding a service operation to a user interface control
- Enabling client-side paging for a service operation. Does not apply to Data Management Service applications.
- Enabling client-side data management for a service operation. Does not apply to Data Management Service applications.

Service classes

When you connect to a service, Flash Builder generates an ActionScript class file that provides access to the service operations.

Flash Builder bases the name of the generated class file on the name you provided for the service in the service wizard. By default, Flash Builder places this class in a package under src. The name of the package is based on the service name. For example, Flash Builder generates the following ActionScript classes for an EmployeeService class.

```
- project
  |
  - src
    |
    + (default package)
    |
    - services.employeeservice
    |
    - _Super_EmployeeService.as
    |
    - EmployeeService.as
```

The super class contains the implementation for the EmployeeService class.

Never edit the super class, which is a generated class. Modifications you make to the super class can be overwritten. Any changes you make to the implementation might result in undefined behavior.

In this example, use EmployeeService.as to extend the generated super class and add your implementation.

Classes for custom data types

Many remote data services provide server-side typing. These services return complex data as a custom data type.

For remote data services that do not return typed data, Flash Builder provides client-side typing. With client-side typing, you use the Flash Builder Configure Operation Return Type dialog to define and configure the data type for complex data returned by the service. For example, for a service that returns employee database records, you define and configure an Employee data type.

Flash Builder generates an ActionScript class for the implementation of each custom data type returned by the service. Flash Builder uses this class to create value objects, which it then uses to access data from the remote service.

For example, Flash Builder generates the following ActionScript classes for an Employee class:

```
- project
  |
  - src
    |
    + (default package)
    |
    - services.employeeservice
    |
    - _Super_Employee.as
    |
    - Employee.as
```

The superclass contains the implementation for the Employee custom data type.

Never edit the superclass, which is a generated class. Modifications you make to the superclass can be overwritten. Any changes you make to the implementation might result in undefined behavior.

In this example, use Employee.as to extend the generated superclass and add your implementation.

Note: Applications that use generated value objects but not generated service wrappers, should call the object's `_initRemoteClassAlias()` method before retrieving any instances of the object from the server, if there are no other references to the object prior to the retrieval.

Binding a service operation to a user interface control

“[Binding service operations to controls](#)” on page 373 shows how you can bind data returned from service operations to a user interface control. When you bind a service operation to a control, Flash Builder generates the following code when you use Flex SDK 4. For Flex SDK 3.4, the same functionality is achieved using Flex SDK 3.4 syntax.

- Declarations tag containing a CallResponder and service tag
- Event handler for calling the service call
- Data binding between the control and the data returned from the operation

Note: The Declarations tag and CallResponder object do not apply to Flex SDK 3.4.

Declarations tag

A Declarations tag is an MXML element that declares non-default, non-visual properties of the current class. When binding a service operation to a user interface, Flash Builder generates a Declarations tag containing a CallResponder and a service tag. The CallResponder and generated service class are properties of the container element, which is usually the Application tag.

The following example shows a Declarations tag providing access to a remote EmployeeService:

```
<fx:Declarations>
    <s:CallResponder id="getAllItemsResult"/>
    <employeesvc:EmployeeSvc id="employeeSvc" destination="ColdFusion"
        endpoint="http://localhost:8500/flex2gateway/"
        fault="Alert.show(event.fault.faultString)" showBusyCursor="true"
        source="EmployeeService.EmployeeSvc"/>
</fx:Declarations>
```

Call Responder

A CallResponder manages results for calls made to a service. It contains a token property that is set to the Async token returned by a service call. The CallResponder also contains a lastResult property, which is set to the last successful result from the service call. You add event handlers to the CallResponder to provide access to the data returned through the lastResult property.

When Flash Builder generates a CallResponder, it generates an id property based on the name of the service operation to which it is bound. The following code sample shows CallResponders for two operations of an EmployeeService. The getAllItems operation is bound to the creationComplete event handler for a DataGrid control. The delete operation is bound to the selected item in the DataGrid control. The DataGrid control displays the items retrieved from the getAllItems service call immediately after it is created. The Delete Item Button control removes the selected record in the DataGrid control from the database.

```
<fx:Declarations>
    <s:CallResponder id="getAllItemsResult"/>
    <employeeservice:EmployeeService id="employeeService" destination="ColdFusion"
        endpoint="http://localhost:8500/flex2gateway/"
        fault="Alert.show(event.fault.faultString)"
        showBusyCursor="true" source="CodeGenCF.CodeGenSvc"/>
    <s:CallResponder id="deleteItemResult"/>
</fx:Declarations>
<mx:DataGrid id="dg" editable="true"
    creationComplete="getAllItemsResult.token =
        employeeService.getAllItems() "dataProvider="{getAllItemsResult.lastResult}">
    <mx:columns>
        <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>
        <mx:DataGridColumn headerText="last_name" dataField="last_name"/>
        <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>
    </mx:columns>
</mx:DataGrid>
<s:Button label="Delete Item"
    click="deleteItemResult.token = employeeService.deleteItem(dg.selectedItem.emp_no) "/>
```

Event handlers

When you bind a service operation to a user interface component, Flash Builder generates an inline event handler for the CallResponder. The event handler manages the results of the operation. You can also create an event handler in an ActionScript code block, and reference that event handler from a property of a user interface component.

Typically, you populate controls such as Lists and DataGrids with data returned from a service. Flash Builder, by default, generates a creationComplete event handler for the control that executes immediately after the control is created. For other controls, Flash Builder generates a handler for the control's default event. For example, for a Button, Flash Builder generates an event for the Button's click event.

Flash Builder generates inline event handlers. The ActionScript code for the event handler is set directly in the event property of the control. The following example shows the generated creation complete event handler for a DataGrid control:

```
<mx:DataGrid id="dg" editable="true"
    creationComplete="getAllItemsResult.token = employeeService.getAllItems() "
    dataProvider="{getAllItemsResult.lastResult}">
    <mx:columns>
        <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>
        <mx:DataGridColumn headerText="last_name" dataField="last_name"/>
        <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>
    </mx:columns>
</mx:DataGrid>
```

You can also create event handlers in ActionScript blocks and reference the event handler from the event property of the control. The following example shows the same event handler for a DataGrid control implemented in an ActionScript block.

```
<mx:Script>
<! [CDATA[
    import mx.events.FlexEvent;
    import mx.controls.Alert;

    protected function dg_creationCompleteHandler(event:FlexEvent):void
    {
        getAllItemsResult.token = employeeService.getAllItems();
    }
]]>
</mx:Script>

. .
<mx:DataGrid id="dg"
    creationComplete="dg_creationCompleteHandler(event)"
    dataProvider="{getAllItemsResult.lastResult}">
<mx:columns>
    <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>
    <mx:DataGridColumn headerText="last_name" dataField="last_name"/>
    <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>
</mx:columns>
</mx:DataGrid>
```

You can also generate event handlers for controls that respond to user events, such as Buttons. The following example shows a generated event handler for a Button that populates a DataGrid control:

```
<mx:Button label="Delete Item" click="deleteItemResult.token =
    employeeService.deleteItem(dg.selectedItem.emp_no)"/>
```

You can also implement the event handler for the Button in an ActionScript block:

```
<mx:Script>
<! [CDATA[
    import mx.controls.Alert;

    protected function delete_button_clickHandler(event:MouseEvent):void
    {
        deleteItemResult.token =
            employeeService.deleteItem(dg.selectedItem.emp_no);
    }
]]>
</mx:Script>

. .
<mx:Button label="Delete Item" id="delete_button"
    click="delete_button_clickHandler(event)"/>
```

Data binding

Flash Builder generates code that binds the data returned from a service operation to the user interface control that displays the data. The following example code that Flash Builder generates to populate a DataGrid control. The getAllItems operation returns a set of employee records for the custom data type, Employee.

The `dataProvider` property of the DataGrid control is bound to the results stored in the CallResponder, `getAllItemsResult`. Flash Builder automatically updates the DataGrid control with `DataGridColumn`s corresponding to each field returned for an Employee record. The `headerText` and `dataField` properties of each column are set according to the data returned in an Employee record.

```
<mx:DataGrid creationComplete="datagrid1_creationCompleteHandler(event)"  
    dataProvider="{getAllItemsResult.lastResult}" editable="true">  
    <mx:columns>  
        <mx:DataGridColumn headerText="gender" dataField="gender"/>  
        <mx:DataGridColumn headerText="emp_no" dataField="emp_no"/>  
        <mx:DataGridColumn headerText="birth_date" dataField="birth_date"/>  
        <mx:DataGridColumn headerText="last_name" dataField="last_name"/>  
        <mx:DataGridColumn headerText="hire_date" dataField="hire_date"/>  
        <mx:DataGridColumn headerText="first_name" dataField="first_name"/>  
    </mx:columns>  
</mx:DataGrid>
```

Enabling client-side paging for a service operation

Note: This content does not apply the Flex applications that use the Data Management Service in LiveCycle Data Services. The Data Management Service provides server-side data management. This content describes client-side data management and paging for RPC services.

When you enable paging, Flash Builder modifies the implementation of the generated service. When you populate a data control (such as a DataGrid or a List) with paged data, Flash Builder determines the number of records visible in the data control and the total number of records in the database. Flash Builder provides these values as arguments to the service operation that you used to implement paging.

You do not have to modify any client application code after paging is enabled.

See “[Enabling paging](#)” on page 369 for more information.

Enabling client-side data management for a service

Note: This content does not apply the Flex applications that use the Data Management Service in LiveCycle Data Services. The Data Management Service provides server-side data management. This content describes client-side data management and paging for RPC services.

In Flash Builder, data management is the synchronization of a set of updates to data on the server. You can enable data management for custom data types returned from the service. With data management enabled, you can modify one or more items in a client application without making any updates to the server. You can then commit all the changes to the server with one operation. You can also revert the modifications without updating any data on the server.

“[Enabling client-side data management](#)” on page 370 shows how to implement this feature.

When you enable data management, Flash Builder modifies the implementation of the generated service class and the generated class for custom data types. Flash Builder creates a `DataManager` to implement this functionality.

When you call service operations for a managed data type, the changes are reflected in the client application. However, data on the server is not updated until you call the `DataManager`’s `commit()` method. The `DataManager` also provides a `revertChanges()` method that restores the data displayed in the client application to the values retrieved from the server before the last `commit()` call.

To access the `commit()` and `revertChanges()` methods for a managed type, first get the `DataManager` for the type, and then call the methods. For example, if you have an instance of `employeeService` and enabled data management for the Employee data type, then you would do the following:

```
employeeService.getDataManager (employeeService.DATA_MANAGER_EMPLOYEE).revertChanges();  
employeeService.getDataManager (employeeService.DATA_MANAGER_EMPLOYEE).commit();
```

You can also call the `commit()` method directly from the `employeeService` instance. Calling the `commit()` method directly from the service instance commits all changes for all managed data types.

```
employeeService.commit();
```

Note: You cannot call `revertChanges()` directly from the service instance.

Using server-side logging with the Model Assembler

As an extension of the Hibernate Assembler, the Model Assembler passes back any errors as is. The Model Assembler provides additional validation on instances and throws errors when validation fails. To log messages from the Model Assembler, use the Service.Data.Fiber logging filter pattern in the logging section of the services-config.xml file. For additional information from the Hibernate Assembler, use the Service.Data.Hibernate filter pattern. For information about server-side logging, see “[Logging](#)” on page 420.

Setting Hibernate properties for a model in a Hibernate configuration file

The Model Assembler lets you set Hibernate properties in an annotation by prefixing the Hibernate property name with `hibernate`. You can also set Hibernate properties for a specific model by creating a file with the property settings.

The file must use the following naming convention:

```
modelName.hibernate.cfg.xml
```

where `modelName` is the name of the model. You must create the file in the WEB-INF/classes directory of your LiveCycle Data Services server.

The WEB-INF/classes directory can contain other Hibernate configuration files, such as the global configuration file, `hibernate.cfg.xml`. The Model Assembler only searches for the configuration file for the model; it does not use the global configuration file.

More Help topics

[“Hibernate configuration files” on page 278](#)

Configuring the model deployment service

The model deployment service deploys models when the LiveCycle Data Services server starts up. By default, the service uses the WEB-INF/datamodel directory in the web application. You can configure an alternative storage directory. If you change the directory, use a full directory path.

The following example shows all possible elements of a model deployment service configuration in the services-config.xml file:

```
<service class="fiber.data.services.ModelDeploymentService" id="model-deploy-service">
<properties>
    <model-persistence-class>
        fiber.data.services.FilePersistence
    </model-persistence-class>
    <model-persistence-directory>
        /path/to/model/directory
    </model-persistence-directory>
</properties>
</service>
```

Note: The default file persistence mechanism is not supported for WAR file (unextracted) web application deployment.

Entity utility

The entity utility, `fiber.runtime.entity.EntityUtility`, provides server-side (Java) access to behavioral aspects of an entity object that represents an application modeling technology entity and to metadata of the relevant entity. Behavioral aspects include derived property calculation, validation with respect to constraints and validations (in styles), and availability of properties inside variants. The property proxy of the entity utility provides APIs for getting and setting properties of an entity object, which enables the entity utility to deal with many varying representations of entity objects.

The Model Assembler uses the entity utility for validation checking before every create and update operation. This happens automatically. Only data properties are sent across the wire to the server so these are the only properties populated on the entity object available to the Model Assembler. Before committing this object to the database, the Model Assembler checks its validity. During validation calculation, the Model Assembler may use other entity utility functions, such as derived property calculation and availability calculations.

If you use the Model Assembler, the entity utility cannot perform validation calculations that involve function and method calls unless implementations of those functions and methods are provided on the server. You can plug in your own function and method implementations.

If you use a custom assembler that extends the Model Assembler, you can use all other functionality of the entity utility. You must use the property proxy of the entity utility to set and get values on entity objects. If you write your own server-side implementation that does not use the Model Assembler or extend it, you can utilize all of the functionality of the entity utility.

The property proxy lets the entity utility handle different representations of entity objects. For example, the Model Assembler uses byte-code-generated Hibernate beans to represent entity instances.

Functions of the entity utility

Validation

The `fiber.runtime.entity.EntityUtility` interface defines validation of constraints and validation expressions in the following methods. For more information, see the Javadoc API documentation for the `EntityUtility` interface.

- A `validate()` method that evaluates all of the constraints of the entity object that is passed in. The method returns `true` only if they are all valid.
- A `validate()` method that evaluates all of the constraints of the entity object passed in except those whose names are included in a set of constraints to exclude. The method returns `true` only if the non-excluded constraints are valid.

- A `validateSubsetOfConstraints()` method that evaluates only the constraints whose names are included in a collection of constraints to evaluate. The method return `true` only if all of the named constraints are valid.

Derived property calculation

The `fiber.runtime.entity.EntityUtility` interface defines several versions of the `populateDerivedProperties()` methods for calculating the derived properties of an entity object. Consumers can perform the following operations:

- Pass in an entity object and a property name to request a single derived property value.
- Pass in in and out entity objects and request that all derived properties be populated on the out object using data values from the in object. The two parameters can be the same object. There is a convenience API that takes a single entity object parameter and assumes it to be both the in and out entity object.
- Pass in in and out entity objects as above with an additional argument specifying the set of derived properties to be calculated.

The two options directly above support a boolean parameter that instructs the entity utility to use derived property values that are already set on the out object during calculation of other derived properties.

The set of derived properties to calculate and boolean reuse parameters let consumers improve performance by instructing the entity utility to calculate only properties that they are interested in and by reusing derived property values that may have been calculated previously and are known not to have changed. This is important for entities whose derived properties involve expensive calculations possibly due to external calls.

Note: *These methods also apply to constraints, which are considered to be derived properties. Also note that values of derived properties calculated and set on the out object during an invocation are reused and not recalculated if the calculation of another derived property refers to a previously calculated value*

Property availability

The entity utility provides the following methods that determine property availability:

- The `isPropertyAvailable()` method takes a property name and an entity object and returns `true` if the property is currently available with respect to variant state.
- The `getAvailableProperties()` method takes an entity object and returns an iterator over the set of names of all of the currently available properties with respect to variant state.

If the entity utility is requested to calculate the value of a derived property inside a variant, it does so without testing the availability of that property. It is up to the consumer to do test the availability of the property and determine whether the value is meaningful.

Structural validity

The value of a property in a Java instance that represents an entity instance is structurally valid if its Java type matches the type declared in the model, and the property is marked as required, its value is non-null. The entity utility checks structural validity before starting any validate and derived property calculations. The following two methods are also exposed for general consumption:

- The `validatePropertyStructure()` method takes an entity object instance and a property name and performs structural validation for a single property.
- The `isStructurallyValid()` method takes an entity object instance and performs validation for all of its properties.

Metadata

The entity utility provides the following metadata-related methods:

- The `getAvailableProperties()` method takes an entity object instance and gets the names of all currently available properties.
- The `getIdentityMap()` method takes an entity object instance and gets an identity map for it.

Handling missing associations

The expression of a derived property can involve a calculation that references a property of an associated entity. For example, a Person entity could contain a property called `maidenName` whose expression is set to `father.lastName` where `father` is an association of type Person.

Consider a request to calculate the `maidenName` property when the passed in Person entity object does not have its `father` property set. This could be an explicit request to calculate all derived properties, or it could be part of validation because the property appears in an expression of a constraint. The default behavior is fail to calculate maiden name. This is because there is no way to discover the value of the `father` property. However, there are cases where it is useful to provide the entity utility with a custom plug-in that helps it gain access to missing associated properties. An example of such a case is the Model Assembler.

Because the Data Management Service supports lazy associations (only identities populated) and load-on-demand associations (not populated at all), the Model Assembler can receive entity objects with unpopulated associations. If the Model Assembler must perform validation and a constraint of an entity object contains a reference to an unpopulated association, the entity utility uses a customizable entity object retriever property that its consumers initialize with their own implementation of the entity object retriever interface. This interface defines the following methods:

- The `getObject()` method takes an entity and an identity map as input parameters. It returns an instance of that entity with the specified `id`.
- The `getProperty()` method takes an entity, an identity map, and a property name as input parameters. It returns the value of the property for an instance of the entity with the specified `id`.

When the entity utility attempts to perform a calculation and determines that an entity object does not provide a property necessary to complete the calculation, it invokes either the `getObject()` method or the `getProperty()` method on its associated entity object retriever. If the missing property is an association of an entity object that it does have, it invokes the `getProperty()` method. If the missing property is a data property of an entity object that it does have, it assumes that entity object is skeletal (only has `id` properties populated) and it invokes the `getObject()` method for that entity object.

Handling method and function calls

Derived properties, including constraints, can reference method and function calls. The entity utility does not immediately know how to evaluate these invocations. Therefore, it allows consumers to plug in implementations for such invocations using custom implementations of two interfaces described below. Consider the following model:

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <service name="CreditCheck">
        <annotation name="ActionScriptGeneration">
            <item name="ServiceType">WebService</item>
            <item name="WSDL">http://10.60.144.67:8080/axis/services/echo?wsdl</item>
            <item name="WSDL-service">CreditScoreImplService</item>
        </annotation>
        <function name="getCreditScore" arguments="SSN:string" return-type="integer"/>
    </service>
    <service name="BackgroundCheck">
        <function name="getBackgroundScore" arguments="SSN:string" return-type="integer"/>
    </service>

    <entity name="Application" persistent="true">
        <id name="applicantSSN" type="string"/>
        <method name="loansTotalPlusExtra" arguments="extra:integer"
               return-type="integer"/>
        <property name="loans" type="Loan[]"/>
        <property name="creditScore" expr="CreditCheck.getCreditScore(applicantSSN)"/>
        <property name="backgroundScore"
                  expr="BackgroundCheck.getBackgroundScore(applicantSSN)"/>
        <constraint name="minimumCreditScore" expr="creditScore < 600"/>
        <constraint name="minimumBackgroundScore" expr="backgroundScore < 100"/>
        <constraint name="notTooMuchInLoans" expr="loansTotalPlusExtra(250) < 1000"/>
    </entity>

    <entity name="Loan">
        <property name="amount" type="integer"/>
    </entity>
</model>
```

The `creditScore` property, and therefore the `minimumCreditScore` constraint, depend on the `CreditCheck` service. The `backgroundScore` property, and therefore the `minimumBackgroundScore` constraint, depend on the `BackgroundCheck` service. The `notTooMuchInLoans` constraint depends on the `loansTotalPlusExtra` method.

In the case of the ActionScript code generator, the generated ActionScript code provides stubs for implementations of the `BackgroundCheck` service and the `loansTotalPlusExtra` method, and it is up to the user to provide those implementations. If the user did not provide the implementations, the entity utility would throw an "implementation not provided" error whenever any access to properties that depend on such functions or methods is made.

Given the proper annotations, the ActionScript code generator creates the necessary Flex-SDK-based code to implement the `CreditCard` service. From the perspective of the ActionScript code generator, there are two types of function and method references:

- References where user-provided custom behavior is required: methods and custom services
- References where the Flex SDK and annotations are used to generate proper code: `WebService`, `RemoteObject`, and `HTTPMultiService` services

On the server side, the following interfaces in the `fiber.runtime.entity` package let Java-based entity utility consumers provide custom implementations for services and functions:

- The `ModelService` interface lets users provide an implementation for the functions of a single service using a single entry point for that service.
- The `AllModelServices` interface lets users provide implementations for all services defined in a model using a single entry point for that model.

- The EntityMethods interface lets users provide implementations for an entity's methods using a single entry point for that entity.

You can set implementations of the ModelService and AllModelServices classes on an entity utility factory or directly on an entity utility instance. In the case of an entity utility factory, the implementations are passed to every entity utility instance that the factory creates as long as at least one property of the entity corresponding to that instance depends on a function of that service.

You can only set implementations of the EntityMethods interface on entity utility instances directly. If both an implementation of the AllModelServices interface and one of the ModelService interface are set for a particular service, the more specific ModelService implementation is invoked.

The code in the following example shows potential implementations of the ModelService, AllModelServices, and EntityMethods interfaces for the CreditCheck service and the loansTotalPlusExtra method.

ModelService implementation

```
public class CreditCheckServerSideImpl implements ModelService
{
    public Object evaluate(String functionName, Map<String, Object> arguments)
    {
        if ("getCreditScore".equals(functionName))
        {
            String ssn = (String)arguments.get("SSN");

            Object result = null; // perform Web Service invocation in Java and get result

            return result;
        }
        else
        {
            return null;
        }
    }
}
```

AllModelServices implementation

```
public class MortgageModelServicesImpl implements AllModelServices
{
    private PropertyUtility propertyUtility;
    public Object evaluate(Model model, String serviceName, String functionName,
                          Map<String, Object> arguments, PropertyUtility propertyUtility)
    {
        InvocationRequest request = new InvocationRequestImpl();
        request.setServiceName(serviceName);
        request.setOperationName(functionName);
        request.setSynchronous(true);
        request.setInputParameters(arguments);
        InvocationResponse response =
        DSCManagerImpl.getInstance().getDSContainer().getServiceEngine().invoke(request);
        return response.getOutputParameters();
    }
    public void setPropertyUtility(PropertyUtility propertyUtility)
    {
        this.propertyUtility = propertyUtility;
    }
}
```

EntityMethods implementation

```
public class ApplicationMethodsImpl implements EntityMethods
{
    private propertyUtility propertyUtility;
    public Object evaluate(String methodName, Object valueObject,
        Map<String, Object> arguments)
    {
        if ("loansTotalPlusExtra".equals(methodName))
        {
            Integer extra = (Integer)arguments.get("extra");
            Object[] loans = (Object []) propertyUtility.getValue(valueObject, "loans");

            int totalAmount = extra.intValue();

            for (int a=0 ; a < loans.length; a++)
            {
                Integer loanAmount = (Integer) propertyUtility.getValue(loans[a], "amount");
                totalAmount = totalAmount + loanAmount;
            }

            return Integer.valueOf(totalAmount);
        }
        else
        {
            return null;
        }
    }
    public void setpropertyUtility(propertyUtility propertyUtility)
    {
        this.propertyUtility = propertyUtility;
    }
}
```

You register the custom server-side implementations with the entity utility factory and the service itself, as the following example shows:

```
EntityUtilityFactory factory = new EntityUtilityFactory(model, buildContext);
factory.setModelServiceImpl("CreditCheck", new CreditCheckServerSideImpl());
// or factory.setAllServicesImpl(new MortgageModelServicesImpl());

Entity entity = model.getEntity("Application");
service = concreteFactory.getEntityUtility(entity);
service.setEntityMethodsImpl(new ApplicationMethodsImpl());
```

You specify the custom server-side implementations in annotations in the associated model file. When the factory is initialized for a given model, it instantiates the custom implementations and sets them up for use if they are available in the current class loader.

For a model with the following annotations, each entity utility that handles the calculation of derived properties whose values depend on functions of the CreditCheck service or methods of the Application entity, calls into these implementations for the appropriate part of the calculation.

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <service name="CreditCheck">
        <annotation name="ServerProperties"
implementation="myPackage.CreditCheckServerSideImpl"/>
        ...
    </service>
    <entity name="Application" persistent="true">
        <annotation name="ServerProperties" implementation="myPackage.ApplicationMethodsImpl"/>
        ...
    </entity>
</model>
```

Property proxy and utility

Property proxy

The property proxy interface, `fiber.runtime.proxy.PropertyProxy`, exposes APIs for getting and setting properties of an entity object, allowing the entity utility to deal with many varying representations of entity objects. An instance of a property proxy wraps around the underlying representation of an entity object and lets consumers uniformly access its properties. The `getValue()`, `setValue()`, and `getInstance()` methods of the `PropertyProxy` interface provide this functionality. For more information, see the Javadoc API documentation.

Property utility

The property utility interface, `fiber.runtime.property.PropertyProxy`, supports varying custom representations of entity objects. This interface is very similar to the `PropertyProxy` interface but instead of wrapping around an entity object instance, all of its methods take an entity object instance as a parameter. For more information, see the Javadoc API documentation.

Entity utility usage of property proxy and utility

The entity utility exposes a property utility setter that lets its consumers associate an implementation of the `PropertyUtility` interface with the service. Whenever the entity utility needs to set or get a property of an entity object that has been passed to it, it checks if the instance implements the `PropertyProxy` interface. If it does, the entity utility uses the methods of the `PropertyProxy` interface to access property values. If it does not, the entity utility passes the instance to the `PropertyUtility` implementation registered with it and requests that it perform that work for it. This scheme enables the entity utility to handle both strategies, which gives its consumers the option of wrapping instances in a property proxy before passing them to the service or registering a property utility implementation with the service and passing unwrapped instances to the service.

Creating an entity utility

Consumers of the entity utility rely on the `EntityUtilityFactory` class for entity utility instance creation. The constructor of the factory itself requires model and build context arguments. It is expected that consumers have resolved the model before passing it to the entity utility factory. The build context is used for any issues encountered during the building of the entity utility as well as its execution.

After a consumer creates a factory, the consumer can use it to create entity utility instances for particular entities of the model corresponding to the factory. To do so, the consumer invokes the `getEntityUtility()` method of the factory passing in a reference to the entity of interest. For more information, see the Javadoc API documentation.

Note: The `FiberAssembler` class defines a member variable called `entityUtilityFactory` that you can use in your customized assemblers. For information on customizing assemblers, see “[Customizing server-side functionality](#)” on page 348.

Chapter 8: Edge Server

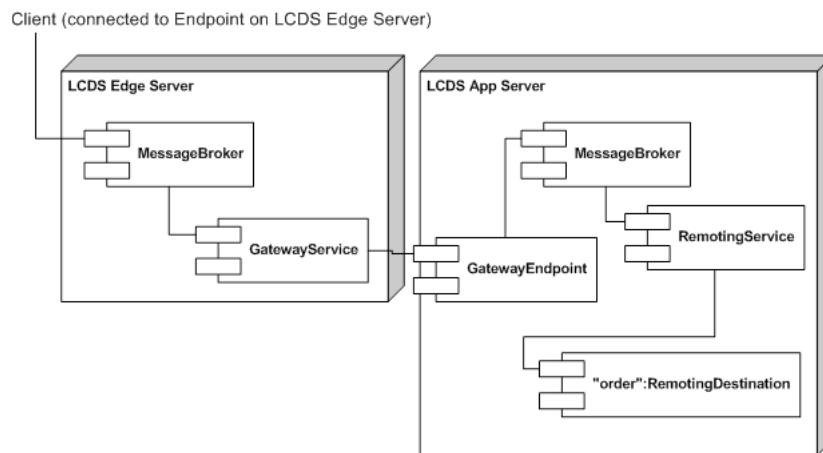
Note: The Edge Server is not available in BlazeDS.

The Edge Server is a LiveCycle Data Services server specially configured and deployed in an organization's demilitarized zone (DMZ), which is the subnetwork that contains and exposes an organization's external services to the Internet. It provides secure and scalable real-time and near-real-time connectivity across the DMZ. The Edge Server forwards authenticated client requests and messages that require sensitive handling, such as database inserts, updates, and deletes, to a LiveCycle Data Services server in the secure application tier (internal network) behind the DMZ.

The Edge Server also lets you optimize application performance by configuring destinations to operate at the Edge Server tier instead of in a LiveCycle Data Services server behind the DMZ. This feature is useful for applications that use high-throughput read-only auto-synced Data Management Service destinations or subscribe-only Message Service destinations. These destinations can connect directly to data sources from within the DMZ rather than connect indirectly through a backing LiveCycle Data Services server. This capability removes network hops and lowers the average latency for pushing updates through these destinations to subscribed clients.

Connecting an Edge Server to a server in the application tier

To connect an Edge Server in the DMZ to a LiveCycle Data Services server in the secure application tier, the Edge Server defines a gateway service and the LiveCycle Data Services server defines a gateway endpoint. The gateway service interacts with the gateway endpoint and forwards messages and requests from clients back to the application tier for secure processing. The following illustration shows a Flex client that makes a remote object call to a Remoting Service destination across a gateway connection:



The gateway endpoint in the secure application tier must listen on a port to which the Edge Server can establish gateway connections on behalf of clients. Because this endpoint is not directly exposed to clients, its configuration differs from the typical client-facing channel/endpoint definition. The following example shows a channel definition that uses the gateway endpoint:

```
<channel-definition id="gateway-endpoint" server-only="true">
    <endpoint url="amfsocket://10.192.16.58:9807"
        class="flex.messaging.endpoints.GatewayEndpoint" />
</channel-definition>
```

Non-client-facing channel/endpoint definitions differ from client-facing channel/endpoint definitions in the following ways:

- No `class` attribute is specified in the `channel-definition` element. Instead, a new `server-only` attribute indicates that clients do not directly use this channel/endpoint definition. This attribute suppresses the inclusion of this `channel-definition` in SWF files compiled against the `services-config.xml` file.
- Along with the server domain name or IP address and optional port number, the `endpoint url` attribute must specify a protocol. When the channel definition is for a gateway endpoint the protocol value must be `amfsocket`.
- The endpoint class must be of type `flex.messaging.endpoints.GatewayEndpoint`.

The default port number for the `amfsocket` protocol is 9807. You do not need to include the port number value in the value of the `url` attribute. As with the existing NIO-based endpoints, this endpoint instantiates its own internal socket server unless it is configured to use a shared socket server. If additional NIO-based endpoints are defined for the LiveCycle Data Services server, configure them to use a single shared socket server unless they must run at different thread priorities.

The following example shows a channel-definition that uses a shared socket server:

```
<channel-definition id="gateway-endpoint" server-only="true">
    <endpoint url="amfsocket://10.192.16.58"
        class="flex.messaging.endpoints.GatewayEndpoint" />
    <server ref="nio-server" />
</channel-definition>
```

At the edge tier, the Edge Server defines a gateway service that connects to the gateway endpoint on the LiveCycle Data Services server in the secure application tier, as the following example shows:

```
<service id="gateway-to-trading-app" class="flex.messaging.services.GatewayService">
    <properties>
        <gateway-endpoint>
            <urls>
                <url>amfsocket://10.192.16.58</url>
                <!-- optional additional urls to GatewayEndpoints on other
                    app server nodes in the backing cluster -->
            </urls>
        </gateway-endpoint>
    </properties>
</service>
```

In this example, the `gateway-endpoint` configuration points to the gateway endpoint defined at the LiveCycle Data Services server in the application tier. As with the gateway endpoint, the `port` element is optional and defaults to 9807. This service always uses the `amfsocket` protocol. The `urls` element is not optional and must contain at least one `url` subelement. If the backing application is clustered, listing more than one address improves the chances that the Edge Server can connect to a cluster node and establish a running gateway.

The gateway service instantiates an internal NIO-based socket connector component that manages all gateway connections to the remote gateway endpoint. This component has a reasonable set of defaults, but exposes a subset of the configuration properties that the NIO socket server also supports. To override any of these settings, define a `connector` element within the `properties` section of the service definition, as the following example shows:

```
<service id="gateway-to-trading-app" class="flex.messaging.services.GatewayService">
    <properties>
        <gateway-endpoint>
            ...
        </gateway-endpoint>
        <connector>
            <properties>
                <max-connection-count>...
                <connection-buffer-type>...
                <connection-read-buffer-size>...
                <connection-write-buffer-size>...
                <socket-keep-alive-enabled>...
                <socket-oobinline-enabled>...
                <socket-receive-buffer-size>...
                <socket-send-buffer-size>...
                <socket-linger-seconds>...
                <socket-tcp-no-delay-enabled>...
                <socket-traffic-class>...
            </properties>
        </connector>
    </properties>
</service>
```

By default, the gateway service `require-authentication` property is set to `true` and you must call the `login()` method of your `ChannelSet` instance(s) before issuing any call that goes through the edge tier. You can set the `require-authentication` property to `false` to authenticate only in the application tier rather than at the edge tier. However, you must ensure that the destination in the application tier is protected by a security constraint so that it requires authentication.

The following example shows the `require-authentication` property set to `false`:

```
<service id="gateway-to-trading-app" class="flex.messaging.services.GatewayService">
    <properties>
        <gateway-endpoint>
            <require-authentication>false</require-authentication>
            ...
        </gateway-endpoint>
        ...
    </properties>
</service>
```

You can configure the gateway service to require a successful connection to its backing server's gateway endpoint during startup. The default value for this setting is `false`. When it is set to `true`, if a connection to the backing application cannot be established, the Edge Server logs an error and does not start. The following example shows the `require-for-startup` property set to `true`:

```
<service id="gateway-to-trading-app" class="flex.messaging.services.GatewayService">
    <properties>
        <gateway-endpoint>
            <require-for-startup>true</require-for-startup>
            ...
        </gateway-endpoint>
        ...
    </properties>
</service>
```

The Edge Server sets up a gateway connection on behalf of a client in a balanced manner across available nodes when clustering is used in the application tier. The Edge Server periodically pings known nodes in the cluster to determine the load they are under. Using this information, the gateway service keeps its view of the backing application cluster up to date. This allows the service to pin each new gateway connection to the least loaded server.

For additional load balancing functionality, you can write code to run within gateway endpoints at the cluster nodes to calculate the load an instance is under based on static or dynamic runtime metrics. You write a custom class that extends the `flex.messaging.endpoints.LoadCalculator` class and implements the `getCurrentLoad()` method. The `getCurrentLoad()` method returns a double with a decimal value representing the current load the process is under. You register your class in the endpoint configuration, as the following example shows:

```
<channel-definition id="gateway-endpoint" server-only="true">
    <endpoint url="amfsocket://yourserver:9807"
        class="flex.messaging.endpoints.GatewayEndpoint"/>
    <properties>
        <load-calculator class="my.custom.StaticLoadAdjuster">
            <properties>
                <relative-load-value>0.5</relative-load-value>
            </properties>
        </load-calculator>
    </properties>
</channel-definition>
```

More Help topics

[“Configuring the socket server” on page 63](#)

Example application configuration

We use a sample stock trading application here to illustrate configuration across the network tiers. The application supports foreign-exchange currency pair trading.

The Flex client receives a high-throughput tick feed of price changes for currency pairs of interest. The client displays this data graphically and runs algorithms locally to compute moving averages that trigger signals when a profitable trade is possible. If the trader places an order, it is submitted to the server securely and the client receives pushed updates on the order status as it moves through approval and clearance steps.

Destinations hosted on the Edge Server

On the Edge Server is a Data Management Service destination with auto synchronization enabled. This destination uses a custom assembler to connect to a Forex price tick application feed. This destination is running at the edge to minimize latency of pushed updates to clients that connect from the public Internet or from partner networks.

Flex clients perform fills against this destination to track currency pairs of interest. The destination is configured to apply conflation in the event of bursts, and the frequency level has been tuned to maintain reasonable throughput during a burst without overwhelming the client or negatively impacting moving average calculations.

For the insecure connection for the Forex tick application feed, clients connect in order of preference using RTMP, streaming AMF, or long polling AMF. For the secure connection for order placement and subscription for order updates, clients connect using the secure versions of RTMP, streaming AMF, or long polling AMF.

DNS CNAMEs are used to allow a client to establish both an insecure and secure streaming or long-polling connection over HTTP without overrunning the browser's connection limit to a single domain. You use DNS CNAMEs to map multiple domain names to the same IP address. For example, app.edge.com and secure.edge.com can both map to 192.150.18.60. The browser treats app.edge.com and secure.edge.com as separate servers. If your browser had a max connections per server limit of two, you would only be able to have two connections to 192.150.18.60 but you could have two connections to app.edge.com and two to secure.edge.com, both of which would resolve to 192.150.18.60.

Destinations hosted on an application tier server

On the application tier server are two Data Management Service destinations that securely handle receipt of order requests from clients and subscriptions for pushed updates to the status of orders. Both destinations are configured to be reliable.

Additionally, some clients run internally behind the DMZ and Edge Server, so the Data Management Service destination for the Forex price tick application feed is also defined in the application tier for those clients to access. Internal clients connect using RTMP. These clients do not connect over RTMPS because they are in a secure region of the corporate network.

Edge Server configuration

There are two Edge Servers in the DMZ with a load balancer in front of them that handles HTTP sticky sessions. These servers are not used for RTMP because that would interfere with reliable messaging during RTMP reconnects. These two servers are not clustered.

The following example shows the relevant parts of the services-config.xml file for the Edge Servers:

```
<services-config>
    <services>
        <service id="data-service" ...>
            ... default channels are the insecure channels ...
            <destination id="forex-tick" ...>
                ... all the config for assembler and inflation ...
            </destination>
        </service>

        <service id="gateway-to-trading-app" ...>
            <gateway-endpoint>
                <urls>
                    <url>10.132.16.56</url>
                    <url>10.132.16.58</url>
                </urls>
            </gateway-endpoint>
        </service>
    </services>

    <servers>
        <server id="nio-server" .../>
        <server id="secure-nio-server" ...>
            ... SSL config ...
        </server>
    </servers>

    <channels>
        <!-- insecure channels -->
        <channel-definition id="rtmp" ...>
            <endpoint url="rtmp://yourserver" .../>
            <!-- default port of 1935 -->
        </channel-definition>
    </channels>
</services-config>
```

```

<server ref="nio-server"/>
<properties>
    <client-load-balancing>
        <!-- lists all available edge server RTMP
        endpoints statically -->
        <url>rtmp://rtmp1.edge.com</url>
        <url>rtmp://rtmp2.edge.com</url>
    </client-load-balancing>
</properties>
</channel-definition>

<channel-definition id="amf-stream" ...>
    <!-- endpoint url points at the HTTP load balancer.
    and the /forex/amfstream path info routes requests to this webapp and
    endpoint -->
    <endpoint url="http://app.edge.com/forex/amfstream" .../>
    <server ref="nio-server"/>
</channel-definition>

<channel-definition id="amf-long-poll" ...>
    <!-- endpoint url points at the HTTP load balancer,
    and the /forex/amflongpoll path info routes requests
    To this webapp and endpoint -->
    <endpoint url="http://app.edge.com/forex/amflongpoll" .../>
    <server ref="nio-server"/>
    <properties>
        ... long poll settings ...
    </properties>
</channel-definition>
<!-- secure channels -->
... similar to above but RTMPS and https ...
... ids match the above ids but have a "secure-" prefix ...
<channel-definition id="secure-amf-stream" ...>
    <!-- endpoint url points at the HTTP load balancer.
    and the /forex/amfstream path info routes requests to this webapp and
    endpoint -->
    <endpoint url="https://secure.edge.com/forex/secureamfstream" .../>
    <server ref="secure-nio-server"/>
</channel-definition>
</channels>
<flex-client>
    <!-- Enable adaptive throttling for clients -->
    <flex-client-outbound-queue-processor ...>
        <properties>
            <adaptive-frequency>true</adaptive-frequency>
        </properties>
    </flex-client-output-queue-processor>
    <!-- Maintain client state at the server for at least 1 minute after a
    disconnect for reliable reconnects -->
    <timeout-minutes>1</timeout-minutes>
</flex-client>
</services-config>

```

Server configuration in application tier

There are two LiveCycle Data Services application servers in the application tier. These servers do not define any clustered service destinations.

The following example shows the relevant parts of the services-config.xml file for the application tier servers:

```
<services-config>
  <services>
    <service id="data-service" ...>
      ... default channels are the insecure channels ...
      <destination id="forex-tick" ...
        ... all the config for assembler and inflation ...
      </destination>
    </service>
    <service id="remoting-service" ...>
      ... default channels are the secure channels ...
      <destination id="trade-placement" ...
        ... config for remote object class ...
      </destination>
    </service>
    <service id="messaging-service" ...>
      ... Default channels are the secure channels ...
      <destination id="trade-confirmation" ...
        ... config for a custom messaging adapter that plugs back into either JMS or another
              backend messaging system that pushes trade order updates and confirmations ...
      </destination>
    </service>
  </services>
  <servers>
    <server id="nio-server" .../>
    <server id="secure-nio-server" ...>
      ... SSL config ...
    </server>
  </servers>
  <channels>
    <channel-definition id="rtmp" ...>
      ...The endpoint url isn't used by clients when client
          load balancing is defined...
      <endpoint url="rtmp://10.132.16.56" .../> <!-- default port of 1935 -->
      <server ref="nio-server"/>
      <properties>
        <client-load-balancing>
          <!-- lists all available internal app server rtmp endpoints statically -->
          <url>rtmp://10.132.16.56</url>
          <url>rtmp://10.132.16.58</url>
        </client-load-balancing>
      </properties>
    </channel-definition>
  </channels>

```

```
</channel-definition>
...
  NOTE: All channel-definitions that the Edge Server uses must be defined here as well even
if internal clients won't use them. Startup verifies that local service destinations in the
application tier reference a channel definition by ID. That ID needs to match what is running
at the Edge Server so that when a client receives runtime configuration that contains the
service destinations for the application tier, it knows which channel it should connect to the
edge tier in order to access them in the application tier. ...
</channels>
<flex-client>
  <!-- Enable adaptive throttling for clients -->
  <flex-client-outbound-queue-processor ...>
    <properties>
      <adaptive-frequency>true</adaptive-frequency>
    </properties>
  </flex-client-output-queue-processor>
  <!-- Maintain client state at the server for at least 1 minute after a
      disconnect for reliable reconnects -->
  <timeout-minutes>1</timeout-minutes>
</flex-client>
</services-config>
```

Note: You can set the *remote* attribute of a *channel-definition* element to "true" to allow an endpoint to be defined on the server but prevent the endpoint from starting up. This is useful when no internal clients use the endpoint. The endpoint defined on the server but does not start up and listen on the port. This configuration avoids overhead of running an endpoint that is not used. Also, the server does not bind to and expose ports that are not needed.

Creating a merged configuration for client compilation

You must compile the SWF file for external clients against a services-config.xml configuration file that explicitly lists all the service destinations that the gateway service exposes. The *services* section of the services-config.xml file must contain the union of configuration for service destinations from the Edge Server configuration and the configuration of the server in the application tier.

Note: Because this services-config.xml is only used for client compilation, it does not have to include configuration for the gateway service. The gateway service is only used on the server. You also do not need perform this procedure when compiling the SWF file for internal clients. When you compile those SWF files, specify the services-config.xml file of the server in the application tier.

In the Edge Server configuration, you define the endpoints that users outside the firewall use to communicate with the LiveCycle Data Services server in the application tier. Users connect to the Edge Server in the DMZ over the endpoints defined in the Edge Server services-config.xml file. The Edge Server then forwards the messages sent over these connections to the backing server over a gateway connection between the edge and application tier. The following example shows an endpoint defined in the Edge Server configuration:

```
<channel-definition id="rtmp" class="mx.messaging.channels.RTMPChannel">
  <endpoint url="rtmp://edge.adobe.com:1935"
            class="flex.messaging.endpoints.RTMPPEndpoint"/>
</channel-definition>
```

In the configuration for application tier server, you define matching endpoints with the same *id* values as the endpoints defined for the Edge Server. Users inside the firewall can use these endpoints to communicate directly with the server in the application tier. If you do not define these endpoints, requests from the Edge Server fail.

The following example shows an endpoint defined in the application tier server configuration:

```
<channel-definition id="rtmp" class="mx.messaging.channels.RTMPChannel">
    <endpoint url="rtmp://my.local.network:2883"
        class="flex.messaging.endpoints.RTMPEndpoint"/>
</channel-definition>
```

In the configuration for the server in the application tier, you also define the destinations that your application exposes. These destinations are available whether users are connected to the Edge Server or directly to the server in the application tier.

The following example shows a destination configuration in the application tier server configuration:

```
<destination id="MyTopic">
    <properties>
        <network>
            <reliable>true</reliable>
            <cluster ref="default-cluster" />
        </network>
    </properties>
    <security>
        <security-constraint ref="sample-users"/>
    </security>
</destination>
```

To compile an MXML application for users outside the firewall who connect to the Edge Server, create a merged services-config.xml file that contains the endpoints from the Edge Server configuration and the destinations from the application tier server configuration. Create a copy of the Edge Server configuration files and add the destination definitions from the server in the application tier.

The following example shows a merged configuration file that contains the endpoints from an Edge Server and the destinations from a LiveCycle Data Services server in the application tier. MXML files compiled against this configuration file connect to an Edge Server on edge.adobe.com and can use the `MyTopic` messaging destination that exists only in the backing LiveCycle Data Services server in the application tier.

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <services>
        <service class="flex.messaging.services.AdvancedMessagingSupport"
            id="AdvancedMessagingSupport"/>
        <service id="message-service"
            class="flex.messaging.services.MessageService">
            <adapters>
                <adapter-definition id="actionscript"
                    class="flex.messaging.services.messaging.adapters.ActionScriptAdapter"
                    default="true" />
            </adapters>
            <destination id="MyTopic">
                <properties>
                    <network>
                        <reliable>true</reliable>
                        <cluster ref="default-cluster" />
                    </network>
                </properties>
                <security>
                    <security-constraint ref="sample-users"/>
                </security>
            </destination>
        </service>
    </default-channels>
```

```
<channel ref="rtmp"/>
</default-channels>
</services>
<security>
    <login-command class="flex.messaging.security.TomcatLoginCommand"
        server="Tomcat"/>
    <security-constraint id="sample-users">
        <auth-method>Custom</auth-method>
        <roles>
            <role>sampleusers</role>
        </roles>
    </security-constraint>
</security>
<channels>
    <channel-definition id="rtmp" class="mx.messaging.channels.RTMPChannel">
        <endpoint url="rtmp://edge.adobe.com:1935"
            class="flex.messaging.endpoints.RTMPEndpoint"/>
    </channel-definition>
</channels>
</services-config>
```

Edge Server authentication and authorization

A gateway service only routes messages to a backing LiveCycle Data Services application if the sending client has successfully authenticated at the Edge Server. Message passing from the gateway service to the backing application can be further constrained by applying a roles-based security constraint, as the following example shows:

```
<service id="gateway-to-trading-app" class="flex.messaging.services.GatewayService">
    <default-security-constraint ref="traders-and-admins"/>
    <properties>
        <gateway-endpoint>
            ...
        </gateway-endpoint>
    </properties>
</service>
```

In this example, a security constraint with an `id` of `traders-and-admins`, separately defined within the top-level `security` section in `services-config.xml`, is applied to this service. Only clients in the set of roles associated with this security constraint are allowed to interact with the backing application over the gateway service.

Regardless of whether authorization is enforced in the gateway service based on a security constraint in the edge tier, any messages routed to service destinations in the backing application are subject to all configured security constraints there (at both the service and destination level) before the message is processed.

Authentication on the Edge Server works the same as it does on the application tier server. You must configure a login command on the Edge Server just as you do on the application tier server. There is no difference in the way you configure a login command on the Edge Server. For more information about authentication and login commands, see “[Security](#)” on page 429.

Restricting access from the Edge Server with white lists and black lists

In a typical Edge Server deployment, the LiveCycle Data Services server is behind an internal firewall. The only port on the LiveCycle Data Services server exposed to incoming connections is the port that the GatewayEndpoint is listening on. Deploying the Edge Server in a DMZ helps ensure that applications outside the external firewall do not have direct access to the GatewayEndpoint of the LiveCycle Data Services server.

To further lock down the GatewayEndpoint, you can use white list and black list rules on the LiveCycle Data Services socket server used by the GatewayEndpoint. White list and black lists can block connections from IP addresses other than the IP address of the Edge Server.

A white list defines the IP addresses permitted to connect to a LiveCycle Data Services server. When defined, client IPs must satisfy the white list filter to connect.

A black list defines the IP addresses that are restricted from connecting to a LiveCycle Data Services server. The black list takes precedence over the white list if an IP address is a member of both lists.

Configure white lists and black lists in a socket server definitions in the services-config.xml file. You can optionally set these properties directly in a channel definition when you are not sharing the same socket server for more than one endpoint.

The following example uses a white lists to specify the IP addresses that can connect to a LiveCycle Data Services server:

```
<whitelist>
    <ip-address>10.132.64.63</ip-address>
    <ip-address-pattern>240.*</ip-address-pattern>
</whitelist>
```

In this example, you specify an explicit IP address corresponding to an Edge Server, and use a wildcard character to specify a range of IP addresses.

More Help topics

[“Configuring the socket server” on page 63](#)

Connecting Flex clients to an Edge Server

Like client connections to a LiveCycle Data Services server in the internal network, client connections to an Edge Server must be sticky. An HTTP connection between the client and Edge Server can involve many physical TCP sockets over its lifetime. So, clients that connect to the server directly must use the server's unique domain name or IP address. For clients connecting through a load balancer that supports HTTP, configure the load balancer to support sticky sessions, pinning all requests from a given client to the same Edge Server. You can base this configuration on the session cookie AMFSessionId that is used with non-RTMP NIO-based endpoints. Servlet-based endpoints are currently not supported for the Edge Server.

When using an RTMP endpoint, the connection between the client and server is a single, long-lived TCP socket. To support reliable messaging reconnect attempts to the server following a transient network disconnect, configure the client to connect to an Edge Server directly, using its unique domain name or IP address. If TCP connections are routed through a load balancer using a simple TCP pass-through in a round-robin fashion, the load balancer does not consistently reestablish a proxy TCP socket on behalf of the reconnected client to the same Edge Server that the client was previously connected to.

Using a load balancer in combination with RTMP connections is not recommended with applications for which you want to take advantage of reliable messaging. To improve support for client connectivity to a tier of Edge Servers when you cannot use a load balancer, you can define a `client-load-balancing` subelement in the `properties` section of the endpoint's `channel-definition` in the `services-config.xml` file. Client applications that you compile against the `services-config.xml` file use this set of URLs to connect to the server rather than the `url` attribute specified in the channel definition's `endpoint` element. In this case, the `endpoint` URL value is not compiled into the client SWF file. Additionally, when runtime configuration is generated and returned, it contains the `client-load-balancing` list of URLs and not the `endpoint` URL.

Before the client initially connects, it shuffles through the full set of URLs specified in the `client-load-balancing` element. The client assigns one URL at random as the primary URL for its `Channel` object. It assigns the remaining URLs to the `failoverURIs` property on its `Channel` object. This process randomly distributes client connections across available Edge Server instances in the absence of a load balancer. The following example channel definition from a `services-config.xml` file shows a `client-load-balancing` element:

```
<channel-definition id="rtmp" class="mx.messaging.channels.RTMPChannel">
<endpoint url="rtmp://10.132.19.65:1935" class="flex.messaging.endpoints.RTMPEndpoint"/>
<properties>
  <client-load-balancing>
    <curl>rtmp://edge1.adobe.com:1935</curl>
    <curl>rtmp://edge2.adobe.com:1935</curl>
    <curl>rtmp://edge3.adobe.com:1935</curl>
  </client-load-balancing>
</properties>
</channel-definition>
```

In this channel definition, the `url` attribute of the `endpoint` element only defines the local port on which the endpoint binds service connections. Clients compiled against this channel definition attempt to connect to URLs listed in the `client-load-balancing` configuration element in a random fashion.

The Edge Server maintains the following objects:

- A server-side `FlexClient` instance that represents the client
- A `FlexSession` or `FlexSessions` that represent the client's connection to the server; for clients that have more than one connection to the server, the server maintains a `FlexSession` for each connection
- The outbound queue of pushed messages for the client, including optional handling of adaptive throttling based on the run-time read rate for individual clients
- An optional reliable messaging sequence if the client is interacting with any reliable service destinations at either the edge tier or application tier

For a client that interacts with a backing server in the application tier through an Edge Server, client state is tracked at both tiers on a single server in each tier. Additionally, this state must be kept loosely in sync across the tiers. When a `FlexClient` or `FlexSession` is invalidated on one tier due to disconnect, timeout or an explicit call to the `invalidate()` method, it is also invalidated at the other tier. Authentication state, in the form of the principal cached in either the `FlexSession` or `FlexClient` is mirrored from the edge tier back to the application tier.

When a SWF file is deployed on a different server than the Edge Server, meaning that the domain name in the request to load the SWF file is different than the domain name used in HTTP requests that the Edge Server makes, a cross-domain policy file is required to allow the SWF file to make requests. By default, the cross-domain file that the NIO server uses is in the `WEB-INF/flex` directory of the web application. You can configure the location in the NIO server configuration in the `services-config.xml` file; you can configure the server to use a subdirectory of the `WEB-INF/flex` directory, but the path must always be rooted at `WEB-INF/flex`.

Handling missing Java types at the edge tier

NIO endpoints at the edge tier can receive AMF data from a client that contains types that do not exist at the edge tier but do exist at the application tier. The edge tier must pass this data through, preserving all type information supplied by the client.

Object instances in the AMF byte stream are converted to ASObject instances that retain the original typing information. This allows objects to be successfully forwarded over gateway connections regardless of whether the Java type is resident at the edge tier.

Types that implement the `IExternalizable` interface at the client require that the corresponding Java class is available at the edge tier. There is no way to read (or read past) the bytes in the AMF stream for an externalizable type.

JMX management

You use the `acceptGatewaySessions` attribute of the gateway endpoint MBean (`GatewayEndpointControlMBean`) to determine whether the gateway endpoint is accepting gateway connections.

You can use `flipAcceptGatewaySessions` operation to start or stop receiving gateway connections. This allows administrators to disable creation of new gateway sessions on target nodes in the application tier, wait for existing sessions to drain, and then pull the node out of the cluster to update before putting it back online to support rolling deployments.

Chapter 9: Generating PDF documents

The Adobe LiveCycle Data Services PDF generation feature enables you to build an Adobe Flex application that can generate a PDF document that includes images and data collected dynamically on the client and sent to the server.

Note: The PDF generation feature is not available in BlazeDS.

About the PDF generation feature

The PDF generation feature lets you build a Flex application that can generate an Adobe Portable Document Format (PDF) document that includes images and/or data.

You use Adobe® LiveCycle® Designer ES to create an Adobe XML Data Package (XDP) document that contains Adobe XML Forms Architecture (XFA) templates. For more information about XFA, see partners.adobe.com/public/developer/xml/index_arch.html.

In a Flex client application, you use existing Flash and Flex APIs to capture images, and you typically represent data as XML. You use a RemoteObject instance to send data to the server. On the server, you write a custom Java class and expose it as a remote object; this class is used to generate PDF documents based on the images and XML data passed from the Flex client application.

Using the PDF generation feature

There are two approaches to this feature. You can use LiveCycle Designer to create a PDF template that contains a valid XDP document with an XFA template. The PDF file then acts as the shell for the XDP and XFA data and contains the necessary pre-rendered AcroForm for display in Adobe Acrobat 7.

Alternatively, for Acrobat 8 clients only, you can use a raw XDP file; this requires a valid LiveCycle Data Services license.

When creating an XDP document in LiveCycle Designer, you must note the dataset document object model (DOM) that is bound to the XFA template. You use this to design the model that captures data in a Flex client application and is sent to a LiveCycle Data Services remote object. The default structure of the model matches the hierarchy of the named elements in the XFA template and thus relies on the normal binding rules. If you have a different XML structure, you must add explicit bindings in your XFA template to match the new structure.

The remote object destination uses a helper class to load an XDP document and add the received data as XML or an input stream that returns XML as input. The destination may store the PDF document on disk or perhaps in a database and typically returns a response reporting the status of the operation, and, potentially, a URL to the generated media.

Generating XML from a PDF template that contains an XDP document

An easy way to get the document object model of an XDP document is to open the PDF file that contains the XDP document in Adobe Acrobat 8 Professional and then export its form data as an XML file.

Complete the following steps to export form data from a XDP document as XML:

- 1 Open the XDP document you want to use in Acrobat 8 Professional.
- 2 On the menu bar, select Forms > Manage Form Data > Export Data.

3 Save the data as XML to create an XML file that has data model structure of the XDP document.

The following example is a sample XML file that was exported from an XDP document:

```
<?xml version="1.0" encoding="UTF-8"?>
<form1>
    <CheckBox1></CheckBox1>
    <DateTimeField1/>
    <DecimalField1/>
    <DropDownList1/>
    <ImageField1/>
    <ListBox1/>
    <NumericField1/>
    <PasswordField1/>
    <TextField1/>
    <Table1>
        <HeaderRow xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:dataNode="dataGroup"/>
        <Row1 xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:dataNode="dataGroup"/>
        <Row2 xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:dataNode="dataGroup"/>
        <Row3 xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:dataNode="dataGroup"/>
    </Table1>
    <RadioButtonList/>
</form1>
```

Creating an MXML file

Using the XML-based data model from the XDP document, you can create an MXML file that uses a Flex XML data model to map data to the XDP document model and send that data to the server with a RemoteObject.

The following example shows an MXML file for generating a PDF from either a PDF that contains an XDP file with an XFA template, or a raw XDP file:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Style source="styles.css" />
    <mx:Script>
        <![CDATA[
            import flash.net.navigateToURL;
            import mx.controls.Alert;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.graphics.ImageSnapshot;

            [Bindable]
            public var balanceSheetImage:String;

            [Bindable]
            public var earningsImage:String;

            private function generatePDFFromPDF():void {
                generatePDF();
                pdfService.generatePDF(xmlModel);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```
private function generatePDFFromXDP():void {
    generatePDF();
    xdpService.generatePDF(xmlModel);
}

private function generatePDF():void {
    var snapshot:ImageSnapshot =
        ImageSnapshot.captureImage(balanceSheet);
    balanceSheetImage = ImageSnapshot.encodeImageAsBase64(snapshot);

    snapshot = ImageSnapshot.captureImage(earnings);
    earningsImage = ImageSnapshot.encodeImageAsBase64(snapshot);
}

private function resultHandler(event:ResultEvent):void {
    var url:String = event.result as String;
    navigateToURL(new URLRequest(url), "_blank");
}

private function faultHandler(event:FaultEvent):void {
    Alert.show("Fault",event.fault.toString());
}
]]>
</mx:Script>

<mx:XML id="xmlModel">
<CompanyReport>
    <TitleText>{panel.title}</TitleText>
    <OverviewText>{overviewText.text}</OverviewText>
    <BalanceSheetImage>{balanceSheetImage}</BalanceSheetImage>
    <EarningsImage>{earningsImage}</EarningsImage>
</CompanyReport>
</mx:XML>

<mx:RemoteObject id="pdfService"
    destination="PDFService"
    fault="faultHandler(event)"
    result="resultHandler(event)"/>

<mx:RemoteObject id="xdpService"
    destination="XDPService"
    fault="faultHandler(event)"
    result="resultHandler(event)"/>

<mx:Panel id="panel"
    title="Company Inc."
    height="600" width="800"
    paddingTop="10" paddingLeft="10" paddingRight="10"
    styleName="h1">
    <mx:Image id="image"/>

    <mx:VBox styleName="plain" height="100%" width="100%">
        <mx:HBox width="100%">
            <!-- Company Overview -->
            <mx:VBox width="50%" height="100%">
                <mx:Label text="Overview" styleName="h2"/>
                <mx:Text id="overviewText" width="100%" height="100%">
```

```
<mx:text>
    Company Inc. (Company) is primarily involved with the
    production, distribution and sale of widgets.
</mx:text>
</mx:Text>
</mx:VBox>

<!-- Annual Earnings -->
<mx:VBox width="50%" height="100%">
    <Earnings id="earnings" xmlns="*" height="250" width="100%"/>
</mx:VBox>
</mx:HBox>

<!-- Balance Sheet Summary -->
<BalanceSheet id="balanceSheet"
    xmlns="*" height="200" width="100%"/>
<mx:HBox>
    <mx:Button label="Create PDF (from PDF)"
        click="generatePDFFromPDF();"/>
    <mx:Button label="Create PDF (from XDP)"
        click="generatePDFFromXDP();"/>
</mx:HBox>
</mx:VBox>
</mx:Panel>
</mx:Application>
```

The data in the `xmlModel` data model in bold text, which contains bound data, maps directly to fields of the XDP data model. The sample `generatePDFFromPDF()`, `generatePDFFromXDP()`, and `resultHandler()` methods are responsible for sending the data to the remote object on the server to generate the PDF. The next section describes the server-side remote objects.

This example uses the `mx.graphics.ImageSnapshot` class, which lets you take snapshot images of Flex user interface components. The underlying Flash Player API is `flash.display.BitmapData.draw()`. The maximum dimensions that `BitmapData.draw()` can capture is 2880x2880 pixels. By default, `ImageSnapshot` is also limited to this. `ImageSnapshot` has the following additional features:

- Ability to specify an image format encoding (PNG is the default, JPG is the only other implementation).
- For components that extend `mx.core.UIComponent`, calls `UIComponent.prepareToPrint()` and when finished calls `UIComponent.finishPrint()`. This lets you change the appearance of the component for capture; for example, remove selected item highlights.
- Conversion of captured images to Base64-encoded Strings for text-based serialization purposes, such as embedding in XML).
- Simple API to specify a desired resolution in dots per inch (DPI) that works out the underlying matrix required to scale the off-screen capture to a particular resolution.
- Ability to control whether the `ImageSnapshot` class tries to take multiple snapshots to support resolutions higher than 2880x2880 by stitching together several snapshots into one big `ByteArray` representing raw bitmap data before applying the encoding (for example, PNG). However, this is limited because a `ByteArray` can only hold 256 megabytes of data. Total composite image resolution is limited to about 8192x8192. By default, the requested DPI is reduced until it fits inside 2880x2880 to avoid run-time errors.

The maximum DPI allowed when taking a snapshot depends on the dimensions of the component being captured and the on-screen resolution. The scale factor is the requested DPI divided by the on-screen resolution, which is then multiplied by the dimensions of the rectangular bounds of the user interface component being captured.

For example, suppose you have a component that is 400x300 pixels in area, has an on-screen resolution of 96 dpi, and a requested resolution is 300 dpi. The resulting scale factor is $300 / 96 = 3.125$ times. Therefore, the captured image will be 1250x937.5 pixels. Because four bytes per channel (ARGB) is required to store the image as a Bitmap, it requires 4687500 bytes (around 4.5 megabytes). When PNG-encoded, this data is compressed considerably; depending on the complexity of the captured image of the component, the compressed image can be as much as 50 times smaller.

Writing the remote object (Java) class

To create a PDF file on the server based on input from a Flex client application, you create a custom Java class and expose it as a remote object (Remoting Service destination). This class uses the flex.acrobat.pdf.XFAHelper class or the flex.acrobat.pdf.XPDXFAHelper (for raw XPD files) class to convert the input from the Flex client application to a PDF document.

The remote object class in the following example corresponds to the `pdfService` RemoteObject component in the example MXML file. The `generatePDF()` method takes an XML document (the XML data model from the MXML file) and passes it to the XFAHelper class, which generates a PDF by combining the XDP document and the XML document. The PDF document is then placed in an HTTP servlet request URL that can be retrieved by the MXML file's `resultHandler()` method.

```
package flex.samples.pdfgen;

import java.io.IOException;
import javax.servlet.http.HttpServletRequest;
import org.w3c.dom.Document;
import flex.acrobat.pdf.XFAHelper;
import flex.messaging.FlexContext;
import flex.messaging.FlexSession;
import flex.messaging.util.UUIDUtils;

public class PDFService
{
    public PDFService()
    {
    }

    public Object generatePDF(Document dataset) throws IOException
    {
        // Open shell PDF
        String source =
            FlexContext.getServletContext().getRealPath("/pdfgen/company.pdf");
        XFAHelper helper = new XFAHelper();
        helper.open(source);

        // Import XFA dataset
        helper.importDataset(dataset);
    }
}
```

```
// Save new PDF as a byte array in the current session
byte[] bytes = helper.saveToByteArray();
String uuid = UUIDUtils.createUUID(false);
FlexSession session = FlexContext.getFlexSession();
session.setAttribute(uuid, bytes);

// Close any resources
helper.close();

HttpServletRequest req = FlexContext.getHttpRequest();
String contextRoot = "/samples";
if (req != null)
    contextRoot = req.getContextPath();
String r = contextRoot + "/dynamic-pdf?id=" + uuid + "&jsessionid=" +
    "+ session.getId();
System.out.println(r);
return r;
}
}
```

The remote object class in the following example corresponds to the `xdpService` RemoteObject component in the example MXML file. The `generatePDF()` method takes an XML document (the XML data model from the MXML file) and passes it to the `XDPXFAHelper` class, which generates a PDF by combining the raw XDP document and the XML document. The PDF document is then placed in an HTTP servlet request URL that can be retrieved by the MXML file's `resultHandler()` method.

```
package flex.samples.pdfgen;

import java.io.IOException;
import javax.servlet.http.HttpServletRequest;
import org.w3c.dom.Document;
import flex.acrobat.pdf.XDPXFAHelper;
import flex.messaging.FlexContext;
import flex.messaging.FlexSession;
import flex.messaging.util.UUIDUtils;

public class XDPXFAService
{
    public XDPXFAService()
    {
    }

    public Object generatePDF(Document dataset) throws IOException
    {
        // Open shell XDP containing XFA template
        String source =
            FlexContext.getServletContext().getRealPath("/pdfgen/company.xdp");
        XDPXFAHelper helper = new XDPXFAHelper();
        helper.open(source);
```

```
// Import XFA dataset
helper.importDataset(dataset);

// Save new PDF as a byte array in the current session
byte[] bytes = helper.saveToByteArray();
String uuid = UUIDUtils.createUUID(false);
FlexSession session = FlexContext.getFlexSession();
session.setAttribute(uuid, bytes);

// Close any resources
helper.close();

HttpServletRequest req = FlexContext.getHttpRequest();
String contextRoot = "/samples";
if (req != null)
    contextRoot = req.getContextPath();
return contextRoot + "/dynamic-pdf?id=
    "+ uuid + "&jsessionid=" + session.getId();
}
```

Configuring Remoting Service destinations

The following example shows the code for the server-side Remoting Service destinations for the remote objects described in the preceding section. This destinations are defined in the remoting-config.xml file.

```
...
<service id="remoting-service"
    class="flex.messaging.services.RemotingService">

    <destination id="PDFService">
        <properties>
            <source>flex.samples.pdfgen.PDFService</source>
        </properties>
    </destination>
    <!-- Note: XDP based PDF generation requires a valid license. This
    sample destination will not work unless a valid key is registered for
    the fds property in the /WEB-INF/flex/license.properties file.
    -->
    <destination channels="my-amf" id="XDPXFAService">
        <properties>
            <source>flex.samples.pdfgen.XDPXFAService</source>
        </properties>
    </destination>
    ...
</service>
...
```

Chapter 10: Run-time configuration

Run-time configuration provides server APIs that let you create, modify, and delete services, destinations, and adapters dynamically on the server without the need for any configuration files.

About run-time configuration

Run-time configuration provides server-side APIs that let you create and delete data services, adapters, and destinations, which are collectively called components. You can create and modify components even after the server is started.

There are many reasons why you might want to create components dynamically. For example, consider the following use cases:

- You want a separate destination for each doctor's office that uses an application. Instead of manually creating destinations in the configuration files, you want to create them dynamically based on information in a database.
- You want to dynamically create and configure Hibernate data management service destinations on server startup based on settings in Hibernate configuration files.
- You want a configuration application to dynamically create, delete, or modify destinations in response to some user input.

There are two primary ways to perform dynamic configuration. The first way is to use a custom bootstrap service class that the MessageBroker calls to perform configuration when the Adobe LiveCycle Data Services server starts up. This is the preferred way to perform dynamic configuration. The second way is to use a RemoteObject instance in a Flex client to call a remote object (Java class) on the server that performs dynamic configuration.

The Java classes that are configurable are MessageBroker, AbstractService and its subclasses, Destination and its subclasses, and ServiceAdapter and its subclasses. For example, you use the flex.messaging.services.HTTProxyService class to create an HTTP proxy service, the flex.messaging.services.http.HTTProxyAdapter class to create an HTTP proxy adapter, and the flex.messaging.services.http.HTTProxyDestination class to create an HTTP proxy destination. Some properties (such as `id`) cannot be changed when the server is running.

The API documentation for these classes is included in the public LiveCycle Data Services Javadoc documentation.

Configuring components with a bootstrap service

To dynamically configure components at server startup, you create a custom Java class that extends the `flex.messaging.services.AbstractBootstrapService` class and implements the `initialize()` method of the `AbstractBootstrapService` class. You can also implement the `start()`, and `stop()` methods of the `AbstractBootstrapService` class; these methods provide hooks to server startup and shutdown in case you need to do special processing, such as starting or stopping the database as the server starts or stops. The following table describes these methods:

Method	Descriptions
public abstract void initialize(String id, ConfigMap properties)	Called by the MessageBroker after all of the server components are created, but just before they are started. Components that you create in this method are started automatically. Usually, you use this method rather than the start() and stop() methods because you want the components configured before the server starts. The id parameter specifies the ID of the AbstractBootstrapService. The properties parameter specifies the properties for the AbstractBootstrapService.
public abstract void start()	Called by the MessageBroker as server starts. You must manually start components that you create in this method.
public abstract void stop()	Called by the MessageBroker as server stops.

You must register custom bootstrap classes in the services section of the services-config.xml file, as the following example shows. Services are loaded in the order specified. You generally place the static services first, and then place the dynamic services in the order in which you want them to become available.

```
<services>
    <service-include file-path="remoting-config.xml"/>
    <service-include file-path="proxy-config.xml"/>
    <service-include file-path="messaging-config.xml"/>
    <service-include file-path="data-management-config.xml"/>
    <service class="dev.service.MyBootstrapService1" id="bootstrap1"/>
    <service id="bootstrap2" class="my.company.BootstrapService2">
        <!-- Bootstrap services can also have custom properties that can be
            processed in initialize method -->
        <properties>
            <prop1>value1</prop1>
            <prop2>value2</prop2>
        </properties>
    </service>
</services>
```

The following example shows a custom bootstrap class for dynamically creating an Data Management Service destination. The initialize() method creates a service and then creates a destination in that service. The first comment in the code shows the equivalent configuration file elements for statically creating the destination. Notice that in the initialize() method, the service creates the destination (by using AbstractService.createDestination), and instead of calling on Destination.createAdapter(), the service creates the JavaAdapter.

```

package flex.samples.runtimeconfig;

import flex.data.DataDestination;
import flex.data.DataService;
import flex.data.adapters.JavaAdapter;
import flex.data.config.DataNetworkSettings;
import flex.data.config.MetadataSettings;
import flex.messaging.config.ConfigMap;
import flex.messaging.config.ThrottleSettings;
import flex.messaging.services.AbstractBootstrapService;

/**
 * This class creates a destination at run time.
 * This technique provides an alternative to
 * statically defining the destination in data-management-config.xml.
 * The destination created by
 * this class is equivalent to a destination created
 * in data-management-config.xml as follows:
 *
 * <destination id="employee">
 *     <adapter ref="java-dao" />
 *     <properties>
 *         <source>samples.crm.EmployeeAssembler</source>
 *         <scope>application</scope>
 *         <metadata>
 *             <identity property="employeeId"/>
 *         </metadata>
 *         <network>
 *             <subscription-timeout-minutes>20</subscription-timeout-minutes>
 *             <paging enabled="false" pageSize="10" />
 *             <throttle-inbound policy="ERROR" max-frequency="500"/>
 *             <throttle-outbound policy="REPLACE" max-frequency="500"/>
 *         </network>
 *     </properties>
 * </destination>
 * To make this destination available at startup,
 * you need to declare this class in services-config.xml as follows:
 * <service
 *     class="flex.samples.runtimeconfig.EmployeeRuntimeDataDestination"
 *     id="runtime-employee" />
 */
public class EmployeeRuntimeDataDestination extends AbstractBootstrapService {

    private DataService dataService;

    /**
     * This method is called by Data Service when it has been initialized but
     * not started.
     */
    public void initialize(String id, ConfigMap properties) {

        dataService = (DataService) getMessageBroker().getService("data-service");
        DataDestination destination = (DataDestination) dataService.createDestination(id);
        // If omitted, default channel will be used

        destination.addChannel("my-rtmp");
        destination.setSource("flex.samples.crm.employee.EmployeeAssembler");
    }
}

```

```
destination.setScope("application");

MetadataSettings metadata = new MetadataSettings();
metadata.addIdentityPropertyName("employeeId");
destination.setMetadataSettings(metadata);

DataNetworkSettings ns = new DataNetworkSettings();
ns.setSubscriptionTimeoutMinutes(20);
ns.setPagingEnabled(false);
ns.setPageSize(10);
ThrottleSettings ts = new ThrottleSettings();
ts.setInboundPolicy(ThrottleSettings.POLICY_ERROR);
ts.setIncomingClientFrequency(500);
ts.setOutboundPolicy(ThrottleSettings.POLICY_REPLACE);
ts.setOutgoingClientFrequency(500);
ns.setThrottleSettings(ts);
destination.setNetworkSettings(ns);

JavaAdapter adapter = new JavaAdapter();
adapter.setId("runtime-java-dao");
adapter.setManaged(true);
adapter.setDestination(destination);
// Instead of relying on Destination.createAdapter, the service creates
// the adapter for you.
}

/**
 * This method is called by Data Services as it starts up (after
 * initialization).
 */
public void start()
{
}

/**
 * This method is called by Data Services as it shuts down.
 */
public void stop()
{
}
}
```

Note: The resources/config/bootstrapServices folder of the LiveCycle Data Services installation contains sample bootstrap services.

Configuring components with a remote object

You can use a remote object to configure server components from a Flex client at run time. In this case, you write a Java class that calls methods directly on components, and you expose that class as a remote object (Remoting Service destination) that you can call from a RemoteObject in a Flex client application. The component APIs you use are identical to those you use in a bootstrap service, but you do not extend the AbstractBootstrapService class.

The following example shows a Java class that you could expose as a remote object to modify a Message Service destination from a Flex client application:

```
package runtimeconfig.remoteobjects;

/*
 * The purpose of this class is to dynamically change a destination that
 * was created at startup.
 */
import flex.messaging.MessageBroker;
import flex.messaging.MessageDestination;
import flex.messaging.config.NetworkSettings;
import flex.messaging.config.ServerSettings;
import flex.messaging.config.ThrottleSettings;
import flex.messaging.services.MessageService;

public class ROMessageDestination
{
    public ROMessageDestination()
    {

    }

    public String modifyDestination(String id)
    {
        MessageBroker broker = MessageBroker.getMessageBroker(null);
        //Get the service
        MessageService service = (MessageService) broker.getService(
            "message-service");

        MessageDestination msgDest =
            (MessageDestination) service.createDestination(id);
        NetworkSettings ns = new NetworkSettings();
        ns.setSessionTimeout(30);
        ns.setSharedBackend(true);
        ...
        msgDest.start();
        return "Destination " + id + " successfully modified";
    }
}
```

The following example shows an MXML file that uses a RemoteObject component to call the `modifyDestination()` method of an `ROMessageDestination` instance:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="run() ">

    <mx:RemoteObject destination="ROMessageDestination" id="ro"
        fault="handleFault(event)"
        result="handleResult(event)"/>

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.*;
            import mx.rpc.remoting.*;
            import mx.messaging.*;
            import mx.messaging.channels.*;

            public var faultstring:String = "";
            public var noFault:Boolean = true;
            public var result:Object = new Object();
            public var destToModify:String = "MessageDest_runtime";

            private function handleResult(event:ResultEvent):void {
                result = event.result;
                output.text += "-> remoting result: " + event.result + "\n";
            }

            private function handleFault(event:FaultEvent):void {
                //noFault = false;
                faultstring = event.fault.faultString;
                output.text += "-> remoting fault: " + event.fault.faultString +
                    "\n";
            }

            private function run():void {
                ro.modifyDestination(destToModify);
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="output" height="200" percentWidth="80" />
</mx:Application>
```

Using assemblers with run-time configuration

When you use dynamic configuration to configure a destination that uses an assembler that extends the `flex.data.assemblers.AbstractAssembler` class, the assembler must be in the application scope in order to retain the properties that are set dynamically. Otherwise, every time an instance gets recreated (when in session or request scope), property settings are lost. This is applicable when you use the Hibernate assembler, the SQL assembler, or a custom assembler that extends `flex.data.assemblers.AbstractAssembler`.

The following example shows a standard configuration for a destination that uses Hibernate assembler in the application scope:

```
<destination id="HibernatePerson">
    <adapter ref="java-dao" />
    <properties>
        <source>flex.data.assemblers.HibernateAssembler</source>
        <scope>application</scope>
    ...

```

The following example shows the equivalent Java code for a destination that uses the Hibernate assembler when using dynamic configuration:

```
private void createDestination3(Service service) {
    String destinationId = "HibernatePerson";
    DataDestination destination =
        (DataDestination)service.createDestination(destinationId);

    String adapterId = "java-dao";
    destination.createAdapter(adapterId);

    destination.setSource("flex.data.assemblers.HibernateAssembler");
    destination.setScope("application");

    MetadataSettings ms = new MetadataSettings();
    ms.addIdentityPropertyName("id");
    ms.addAssociationSetting(new AssociationSetting("one-to-many", "groups",
        "Group"));

    // Note that these are HibernateAssembler-specific properties and they
    // need to be set on the assembler. These properties are only retained
    // if scope of the destination is application.
    HibernateAssembler assembler =
        (HibernateAssembler)destination.getFactoryInstance().lookup();
    assembler.setUpdateConflictMode("PROPERTY");
    assembler.setDeleteConflictMode("OBJECT");
    assembler.setHibernateEntity("dev.contacts.hibernate.Person");
    assembler.setHibernateConfigFile("hibernate.cfg.xml");
    assembler.setPageQueriesFromDatabase(true);
    assembler.setUseQueryCache(false);
    assembler.setAllowHQLQueries(false);
}
```

Accessing dynamic components with a Flex client application

Using a dynamic destination with a client-side data services component, such as an `HTTPService`, `RemoteObject`, `WebService`, `Producer`, or `Consumer` component, is essentially the same as using a destination configured in the `services-config.xml` file.

It is a best practice to specify a channel set and channel when you use a dynamic destination, and this is required when there are not application-level default channels defined in the `services-config.xml` file.

If you do not specify a channel set and channel when you use a dynamic destination, LiveCycle Data Services attempts to use the default application-level channel assigned in the `default-channels` element in the `services-config.xml` file. If you rely on application-level default channels, you must make sure that those channels are the ones expected by the destination you are using. For example, if the dynamic destination is created without specifying any channels on the server, it implicitly expects connections only over the service-level default channels. This is fine if service-level and application-level channels match, but if service-level default channels are different than application-level default channels, the client cannot contact the destination using the application-level default channels. In that case, a channel set with the proper channel is required on the client and you cannot rely on the application-level default channels.

The following example shows a default channel configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <services>
        ...
        <default-channels>
            <channel ref="my-polling-amf" />
        </default-channels>
        ...
    </services>
    ...
</services-config>
```

You have the following options for adding channels to a `ChannelSet`:

- Create channels on the client, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = new AMFChannel("my-amf",
    "http://servername:8400/messagebroker/amfpolling");
cs.addChannel(pollingAMF);
...
```

- If you have compiled your application with the `services-config.xml` file, use the `ServerConfig.getChannel()` method to retrieve the channel definition, as the following example shows:

```
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = ServerConfig.getChannel("my-amf");
cs.addChannel(pollingAMF);
```

Usually, there is no difference in the result of either of these options, but there are some properties that are set on the channel and used by the client code.

When you create a polling `AMFChannel` instance on the client, call the `enablePolling` method. Next, set the `pollingInterval` property, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = new AMFChannel("my-amf",
    "http://servername:8100/eqa/messagebroker/amfpolling");
pollingAMF.enablePolling();
pollingAMF.pollingInterval = 8000;
cs.addChannel(pollingAMF);
...
```

You do not have to turn on the polling on the client if you define the polling information in the channel definition of the `services-config.xml` file. When the client connects to the server, it automatically gets configured to poll at the interval specified in the `services-config.xml` channel definition.

For components that use clustered destinations, you must define a ChannelSet and set the `clustered` property of the ChannelSet to `true`.

The following example shows MXML code for declaring a RemoteObject component and specifying a ChannelSet and Channel:

```
<RemoteObject id="ro" destination="Dest">
    <mx:channelSet>
        <mx:ChannelSet>
            <mx:channels>
                <mx:AMFChannel id="myAmf"
                    uri="http://myserver:2000/myapp/messagebroker/amf"/>
            </mx:channels>
        </mx:ChannelSet>
    </mx:channelSet>
</RemoteObject>
```

The following example shows equivalent ActionScript code:

```
private function run():void {
    ro = new RemoteObject();
    var cs:ChannelSet = new ChannelSet();
    cs.addChannel(new AMFChannel("myAmf",
        "http://{server.name}:{server.port}/eqa/messagebroker/amf"));
    ro.destination = "RemotingDest_runtime";
    ro.channelSet = cs;
}
```

One slight difference is that when you declare your Channel in MXML, you cannot have the dash (-) character in the value of `id` attribute of the corresponding channel that is defined on the server. For example, you would not be able to use a channel with an `id` value of `message-dest`. This is not an issue when you use ActionScript instead of MXML.

When using the Data Management Service with dynamic DataDestinations in nested DataService associations, you should create a ChannelSet for the topmost DataService object. For example, if you have a nested association of teams, players, and addresses all managed by DataService objects, you would create a ChannelSet for teams (the topmost DataService).

Note: When using the Data Management Service, run-time configuration information is saved to the local cache when present during a save, and is restored during initialization if a connection cannot be established. For more information, see “[Occasionally connected clients](#)” on page 310.

Chapter 11: Administering LiveCycle Data Services applications

Logging

One tool that can help in debugging applications is the logging mechanism. You can perform both client-side and server-side logging. Client-side logging writes log messages from the Flex client to a file on the client computer. Server-side logging writes log messages from the LiveCycle Data Services server to a designated logging target, which can be the log location for the servlet container, System.out, or a custom location.

Client-side logging

For client-side logging, you can directly write messages to the log file, or configure the application to write messages generated by Flex to the log file. The Flash Debug Player has two primary methods of writing messages to a log file:

- The global `trace()` method

The global `trace()` method prints a String to the log file. Messages can contain checkpoint information to signal that your application reached a specific line of code, or the value of a variable.

- Logging API

The logging API, implemented by the `TraceTarget` class, provides a layer of functionality on top of the `trace()` method. For example, you can use the logging API to log debug, error, and warning messages generated by Flex during application execution.

The Flash Debug Player sends logging information to the `flashlog.txt` file. The operating system determines the location of this file, as the following table shows:

Operating system	Log file location
Windows 95/98/ME/2000/XP	C:\Documents and Settings\username\Application Data\Macromedia\Flash Player\Logs
Windows Vista	C:\Users\username\AppData\Roaming\Macromedia\Flash Player\Logs
Mac OS X	/Users/username/Library/Preferences/Macromedia/Flash Player/Logs/
Linux	/home/username/.macromedia/Flash_Player/Logs/

Use settings in the `mm.cfg` text file to configure the Flash Debug Player for logging. If this file does not exist, you can create it when you first configure the Flash Debug Player. The following table shows where to create the `mm.cfg` file for different operating systems:

Operating system	Create file in ...
Mac OS X	/Library/Application Support/Macromedia
Windows 95/98/ME	%HOMEDRIVE%\%HOMEPATH%

Operating system	Create file in ...
Windows 2000/XP	C:\Documents and Settings\username
Windows Vista	C:\Users\username
Linux	/home/username

The mm.cfg file contains many settings that you can use to control logging. The following sample mm.cfg file enables error reporting and trace logging:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

Once logging is enabled, you can call the `trace()` method to write a String to the flashlog.txt file, as the following example shows:

```
trace("Got to checkpoint 1.");
```

To enable the logging of all Flex-generated debug messages to flashlog.txt, insert the following TraceTarget component in your application:

```
<mx:TraceTarget loglevel="2"/>
```

The following example shows an MXML application that uses a TraceTarget component:

```
<?xml version="1.0"?>
<!!-- charts/MXMLTraceTarget.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="initApp();">

    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        import mx.logging.Log;

        [Bindable]
        public var myData:ArrayCollection;

        private function initApp():void {
            Log.addTarget(logTarget);
        }
    ]]></mx:Script>

    <mx:TraceTarget id="logTarget" includeDate="true" includeTime="true"
        includeCategory="true" includeLevel="true">
        <mx:filters>
            <mx:Array>
                <mx:String>mx.rpc.*</mx:String>
                <mx:String>mx.messaging.*</mx:String>
            </mx:Array>
        </mx:filters>
        <!-- 0 is represents the LogEventLevel.ALL constant. -->
        <mx:level>0</mx:level>
    </mx:TraceTarget>

    <!-- HTTPService is in the mx.rpc.http.* package -->
    <mx:HTTPService
```

```
        id="srv"
        url="../assets/data.xml"
        useProxy="false"
        result="myData=ArrayCollection(srv.lastResult.data.result)"
    />

    <mx:LineChart id="chart" dataProvider="{myData}" showDataTips="true">
        <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:LineSeries yField="apple" name="Apple"/>
            <mx:LineSeries yField="orange" name="Orange"/>
            <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
    </mx:LineChart>

    <mx:Button id="b1" click="srv.send();" label="Load Data"/>

</mx:Application>
```

For information about client-side logging, see the Flex documentation set.

Note: For the `HTTPService` object, the `debug` logging setting prints out both the request and the response messages.

Server-side logging

You perform server-side logging for requests to and responses from the server. The following example shows a log message generated by the server:

```
[LCDS] 05/13/2008 14:27:18.842 [ERROR] [Message.General] Exception when invoking service:
(none) with message: Flex Message (flex.messaging.messages.AsyncMessageExt)
    clientId = 348190FC-2308-38D7-EB10-57541CC2440A
    correlationId =
    destination = foo
    messageId = E0BFF004-F697-611B-3C79-E38956BAB21B
    timestamp = 1210703238842
    timeToLive = 0
    body = dsafasdasd: asdasd
    hdr(DSEndpoint) = my-rtmp
    hdr(DSId) = 348190D5-130A-1711-B193-25C4415DFCF5
    hdr(DSValidateEndpoint) = true
exception: flex.messaging.MessageException: No destination with id 'chat' is registered with
any service.
```

You can configure the logging mechanism to specify the following information in the message:

- The type of messages to log, called the *log level*. The available levels include All, Debug, Error, Info, None, and Warn. For example, you can choose to log Error messages, but not Info messages. For more information, see “[Setting the logging level](#)” on page 423.
- The optional String prefixed to every log message. In this example, the String is [LCDS]. For more information, see “[Setting logging properties](#)” on page 424.
- The display of the date and time of the log message. In this example, the message contains the date and time: 05/13/2008 14:27:18.842. For more information, see “[Setting logging properties](#)” on page 424.

- The display of the level of the log message. In this example, the message contains the level: [ERROR]. For more information, see “[Setting logging properties](#)” on page 424.
- The display of the category of the log message. The category provides information about the area of LiveCycle Data Services that generated the message. In this example, the message contains the level: [Message.General]. For more information, see “[Setting logging properties](#)” on page 424.
- The target of the log messages. By default, log messages are written to System.out. For more information, see “[Setting the logging target](#)” on page 424.

Configuring server-side logging

Configure server-side logging in the `logging` section of the `services-config.xml` configuration file. After you edit `services-config.xml`, restart the LiveCycle Data Services server.

The following example shows a configuration that sets the logging level to `Debug`:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[LCDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint.RTMP</pattern>
        </filters>
    </target>
</logging>
```

Setting the logging level

The `level` defines the types of messages written to the log. The following table describes the logging levels:

Logging level	Description
All	Logs all messages.
Debug	Logs debug message. Debug messages indicate internal Flex activities. Select the <code>Debug</code> logging level to include <code>Debug</code> , <code>Info</code> , <code>Warn</code> , and <code>Error</code> messages in your log files.
Error	Logs error messages. Error messages indicate when a critical service is not available or a situation restricts use of the application.
Info	Logs information messages. Information messages indicate general information to the developer or administrator. Select the <code>Info</code> logging level to include <code>Info</code> and <code>Error</code> messages in your log files.
None	No messages are logged.
Warn	Logs warning messages. Warning messages indicate that Flex encountered a problem with the application, but the application does not stop running. Select the <code>Warn</code> logging level to include <code>Warn</code> and <code>Error</code> messages in your log files.

In a production environment, you typically set the logging level to `Warn` to capture both warnings and error messages. If you prefer to ignore warning messages, set the level to `Error` to display only error messages.

Setting the logging target

By default, the server writes log messages to System.out. In the `class` attribute of the `target` element, you can specify `flex.messaging.log.ConsoleTarget` (default) to log messages to the standard output, or the `flex.messaging.log.ServletLogTarget` to log messages to the default logging mechanism for servlets for your application server.

Setting logging properties

The following table describes the logging properties:

Property	Description
<code>includeCategory</code>	Determines whether the log message includes the category. The category provides information about the area of LiveCycle Data Services that generated the message. The default value is <code>false</code> .
<code>includeDate</code>	Determines whether the log message includes the date. The default value is <code>false</code> .
<code>includeLevel</code>	Determines whether the log message includes the log level. The categories are Debug, Error, Info, and Warn. Specifies to include the message category in the logging message. The default value is <code>false</code> .
<code>includeTime</code>	Determines whether the log message includes the time. The default value is <code>false</code> .
<code>filters</code>	Specifies a pattern that defines the categories to log. The category of a log message must match the specified pattern to be written to the log. For more information, see “ Setting a filtering pattern ” on page 424.
<code>prefix</code>	Specifies the String prefixed to log messages. The default value is an empty String.

In the following example, you set the configuration properties to display the category, date, level, time, and set the prefix to [LCDS]:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[LCDS]</prefix>
            <includeDate>true</includeDate>
            <includeTime>true</includeTime>
            <includeLevel>true</includeLevel>
            <includeCategory>true</includeCategory>
        </properties>
    </target>
</logging>
```

Setting a filtering pattern

The `<filters>` property lets you filter log messages based on the message category. If you omit a setting for the `<filters>` property, messages for all categories are written to the log.

The following example shows the first line of log messages from different categories:

```
[LCDS] 05/14/2008 12:52:52.606 [DEBUG] [Endpoint.RTMP] Received command: TCCommand
...
[LCDS] 05/14/2008 12:52:52.606 [DEBUG] [Message.General] Before invoke service: message-service
...
[LCDS] 05/14/2008 12:52:52.606 [DEBUG] [Service.Message] Sending message: Flex Message ...
[LCDS] 05/14/2008 12:52:52.606 [DEBUG] [Message.Timing] After invoke service: message-service;
```

To filter messages so only those messages in the Message.General and Endpoint categories appear, set the `<filters>` property as the following example shows:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[LCDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint.*</pattern>
            <pattern>Message.General</pattern>
        </filters>
    </target>
</logging>
```

Use the wildcard character (*) in the pattern to log messages from more than one category. To see messages for all endpoints, specify a pattern of `Endpoint.*`. To see messages for only an RTMP endpoint, specify a pattern of `Endpoint.RTMP`. To see all messages for all categories, specify a pattern of `*`.

You can use many different patterns as the value of the `pattern` element, such as the following:

Client.*

Client.FlexClient

Client.MessageClient

Configuration

Endpoint.*

Endpoint.General

Endpoint.AMF

Endpoint.NIOAMF

Endpoint.FlexSession

Endpoint.HTTP

Endpoint.NIOHTTP

Endpoint.RTMP

Endpoint.StreamingAMF

Endpoint.StreamingHTTP

Endpoint.Type

Executor

Message.*

Message.General

Message.Command.*

Message.Command.(operation-name) where operation-name is one of subscribe, unsubscribe, poll, poll_interval, client_sync, server_ping, client_ping, cluster_request, login, logout

Message.coldfusion

Message.Data.*

Message.Data.(operation-name) where operation-name is one of create, fill, get, update, delete, batched, multi_batch, transacted, page, count, get_or_create, create_and_sequence, get_sequence_id, association_add, association_remove, fillids, refresh_fill, update_collection

Message.Remoting

Message.RPC

Message.Selector

Message.Timing

Model.*

Model.Deployment

Model.Generation

Model.Configuration

Protocol.*

Protocol.HTTP

Protocol.RTMP

Protocol.RTMPT

Resource

Service.*

Service.Cluster

Service.Data.*

Service.Data.Fiber

Service.Data.General

Service.Data.Hibernate

Service.Data.SQL

Service.Data.Transaction

Service.HTTP

Service.Message

Service.Message.*

Service.Message.JMSService.RemotingSecuritySocketServer.*SocketServer.GeneralSocketServer.ByteBufferManagerSSLStartup.*Startup.MessageBrokerStartup.ServiceStartup.DestinationTimeoutWSRPDataService.coldfusion

For the complete list of filter patterns, see the services-config.xml file in the *install_root/resources/config* directory.

Monitoring and managing services

LiveCycle Data Services uses Java Management Beans (MBeans) to provide run-time monitoring and management of the services configured in the services configuration file. The run-time monitoring and management console is an example of a Flex client application that provides access to the run-time MBeans. The application calls a Remoting Service destination, which is a Java class that makes calls to the MBeans.

About the run-time monitoring and management console

The run-time monitoring and management console is a Flex client application that provides access to the run-time MBeans in the data services web applications running on an application server. The console application calls a Remoting Service destination, which is a Java class that makes calls to the MBeans.

The console is in the ds-console web application; you run it by opening `http://server:port/ds-console` when the web application is running, where `server:port` contains your server and port names.

The tabs in the console provide several different views of run-time data for the data service applications running in the application server. You use the Application combobox to select the web application you want to monitor. You can use a slider control to modify the frequency with which the console polls the server for new data.

The general administration tab provides a hierarchical tree of all the MBeans in the selected web application. Other views target specific types of information. For example, tabs provide server, channel endpoint, and destination information. These tabs include dynamic data graphs for applicable properties.

One log manager tab shows the log categories set in the services-config.xml file and lets you add or remove log categories at run time. You change the log level (debug, info, warn, and so forth) for the log categories. For information about logging, see “[Server-side logging](#)” on page 422.

Note: *The run-time monitoring and management console exposes administrative functionality without authorization checks. Deploy the ds-console web application to the same application server that your Flex web application is deployed to, and lock down the ds-console web application by using J2EE security or some other means to protect access to it. For more information about J2EE security, see your application server documentation and http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security.html.*

MBean creation and registration

The APIs exposed by the run-time MBeans do not affect or interact with the LiveCycle Data Services configuration files. You can use them for operations such as ending a stale connection or monitoring message throttling, but you cannot use them for operations such as registering a new service or altering the settings for an existing server component.

The run-time MBeans are instantiated and registered with the local MBean server by their corresponding managed resource. For example, when a MessageBroker is instantiated, it creates and registers a corresponding MessageBrokerControlMBean with the MBean server. The underlying resource sets the attributes for the run-time MBeans and they are exposed in a read-only manner by the MBean API. In some cases, an MBean can poll its underlying resource for an attribute value.

MBean naming conventions

The run-time MBean model starts at the MessageBrokerControlMBean. You can traverse the model by using attributes on MBeans that reference other MBeans. For example, to access an EndpointControlMBean, you start at the MessageBrokerControlMBean and get the `Endpoints` attribute. This attribute contains the ObjectNames of all endpoints that are registered with the management broker. It lets any management client generate a single `ObjectName` for the root MessageBrokerControlMBean, and from there, navigate to and manage any of the other MBeans in the system without having their `ObjectName`.

The run-time MBean model is organized hierarchically; however, each registered MBean in the system has an MBean ObjectName and can be fetched or queried directly if necessary. The run-time MBeans follow ObjectName conventions to simplify registration, lookup, and querying. An ObjectName for an MBean instance is defined as follows:

```
{domain} : {key}={value} [, {keyN}={valueN}] *
```

You can provide any number of additional key-value pairs to uniquely identify the MBean instance.

All of the run-time MBeans belong to the flex.run-time domain. If an application name is available, it is also included in the domain as follows: `flex.runtime.application-name`.

Each of the run-time MBean ObjectNames contains the following keys:

Key	Description
type	Short type name of the resource managed by the MBean. The MessageBrokerControlMBean manages the flex.messaging.MessageBroker, so its type is MessageBroker.
id	The <code>id</code> value of the resource managed by the MBean. If no <code>id</code> is available on the resource, an <code>id</code> is created according to this strategy: <code>id = {type} + N</code> where <code>N</code> is a numeric increment for instances of this type.

An ObjectName can also contain additional optional keys.

The run-time MBeans are documented in the public LiveCycle Data Services Javadoc documentation. The Javadoc also includes documentation for the `flex.management.jmx.MBeanServerGateway` class, which the run-time monitoring and management console uses as a Remoting Service destination.

Creating a custom MBean for a custom ServiceAdapter class

You can write a custom MBean to expose metrics or management APIs for a custom ServiceAdapter. The following method of the ServiceAdapter class lets you connect your MBean.

```
public void setupAdapterControl(Destination destination);
```

Your custom ServiceAdapter control MBean should extend `flex.management.runtime.messaging.services.ServiceAdapterControl`. Your MBean must implement your custom MBean interface, which in turn must extend `flex.management.runtime.messaging.services.ServiceAdapterControlMBean`. `ServiceAdapterControlMBean` lets you add your custom MBean into the hierarchy of core LiveCycle Data Services MBeans.

The code in the following example shows how to implement a `setupAdapterControl()` method in your custom ServiceAdapter, where `controller` is an instance variable of type `CustomAdapterControl` (your custom MBean implementation class):

```
public void setupAdapterControl(Destination destination) {
    controller = new CustomAdapterControl(getId(), this, destination.getControl());
    controller.register();
    setControl(controller);
}
```

Your custom adapter can update metrics stored in its corresponding MBean by invoking methods or updating properties of `controller`, the MBean instance. Your custom MBean is passed a reference to your custom adapter in its constructor. Access this reference in your custom MBean by using the `protected ServiceAdapter serviceAdapter` instance variable to query its corresponding service adapter for its state, or to invoke methods on it.

Security

LiveCycle Data Services security lets you control access to server-side destinations. A Flex client application may only connect to the server over a secure destination after its credentials have been validated (authentication), and may only perform authorized operations. By default, LiveCycle Data Services uses the security framework of the underlying J2EE application server to support authentication and authorization. However, you can define custom logic to perform authentication and authorization that does not rely on the application server.

Securing LiveCycle Data Services

Authentication is the process by which users validate their identity to a system. *Authorization* is the process of determining what types of activities a user is permitted to perform on a system. After users are authenticated, they can be authorized to access specific resources.

In LiveCycle Data Services, a destination defines the server-side code through which a client connects to the server. You restrict access to a destination by applying a security constraint to the destination.

Security constraints

A security constraint ensures that a user is authenticated before accessing the destination. A security constraint can also require that the user is authorized against roles defined in a user store to determine if the user is a member of a specific role.

You can configure a security constraint to use either basic or custom (application container) authentication. The type of authentication you specify determines the type of response the server sends back to the client when the client attempts to access a secured destination with no credentials or invalid credentials. For basic authentication, the server sends an HTTP 401 error to indicate that authentication is required, which causes a browser challenge in the form of a login dialog box. For custom authentication, the server sends a fault to the client to indicate that authentication is required.

By default, security constraints uses custom authentication.

Login commands

LiveCycle Data Services uses a login command to check credentials and log the user into the application server. The way a J2EE application server implements security is specific to each server type. Therefore, LiveCycle Data Services includes login command implementations for Apache Tomcat, JBoss, Oracle Application Server, BEA WebLogic, IBM WebSphere, and Adobe JRun.

You can use a login command without roles to support authentication only. If you also want to use authorization, link the specified role references to roles that are defined in the user store of your application server.

A login command must implement the `flex.messaging.security.LoginCommand` interface. You can create a custom login command by implementing the `LoginCommand` interface. A custom login command can utilize the underlying J2EE application server, or implement its own validation mechanism.

Classes that implement the `LoginCommand` interface should also implement the `LoginCommandExt` interface in the uncommon scenario where the name stored in the `java.security.Principal` instance that is created upon successful authentication differs from the username passed into the authentication. Implementing the `LoginCommandExt` interface lets the `LoginCommand` instance return the resulting username so that it can be compared to the one stored in the `Principal` instance.

The following code example shows a custom login command class called CustomLoginCommand. This class lets everything through, but it provides an idea of what you need to override. In the `doAuthentication()` and `doAuthorization()` methods, you add custom logic to authenticate with the system. Then, you must configure the login adapter in the `services-config.xml`.

```
package features.test.logincommand;
import java.security.Principal;
import java.util.List;
import javax.servlet.ServletConfig;
import flex.messaging.security.LoginCommand;
public class CustomLoginCommand implements LoginCommand
{
    public Principal doAuthentication(String username, Object credentials)
    {
        // Return a dummy principal without checking the credentials.
        return new CustomPrincipal(username);
    }
    public boolean doAuthorization(Principal principal, List roles)
    {
        // Always return true for now.
        return true;
    }
    public boolean logout(Principal principal)
    {
        // Always return true for now.
        return true;
    }
    public void start(ServletConfig config)
    {
        // No-op.
    }
    public void stop()
    {
        // No-op.
    }
    class CustomPrincipal implements Principal
    {
        private String username;
        public CustomPrincipal(String username)
        {
            this.username = username;
        }
        public String getName()
        {
            return username;
        }
    }
}
```

In the `login-command` element in the `services-config.xml` file, the `server` attribute is set to `"all"` to ensure this login command is used no matter which application server LiveCycle Data Services is deployed on:

```
...
<security>
    <login-command class="features.test.logincommand.CustomLoginCommand" server="all"/>
...

```

For more information about configuring login commands, see “[Configuring security](#)” on page 432.

Secure channels and endpoints

In addition to providing security that lets you authenticate and authorize users, you can also use secure channels and endpoints to implement a secure transport mechanism. For example, the following AMF channel definitions use a nonsecure transport connection:

```
<!-- Non-polling AMF -->
<channel-definition id="samples-amf"
    class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:8100/myapp/messagebroker/amf"
        type="flex.messaging.endpoints.AmfEndpoint"/>
</channel-definition>
```

LiveCycle Data Services provides a secure version of the AMF, HTTP, and RTMP channels and endpoints that transport data over an RTMPS or HTTPS connection. The names of the secure channels and endpoints all start with the string “Secure”. The following example redefines the AMF channel to use a secure transport mechanism:

```
<!-- Non-polling secure AMF -->
<channel-definition id="my-secure-amf"
    class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://{server.name}:9100/dev/messagebroker/amfsecure"
        class="flex.messaging.endpoints.SecureAMFEndpoint"/>
</channel-definition>
```

For more information on using the secure transport mechanism, see “[Channels and endpoints](#)” on page 38.

Secure socket servers

A secure socket server definition requires a digital certificate, and contains child elements for specifying a keystore filename and password. You can create public and private key pairs and self-signed certificates with the Java keytool utility. The user name of self-signed certificates must be set to `localhost` or the IP address on which the RTMPS endpoint is available. For more information, see “[Channels and endpoints](#)” on page 38.

Setting up security constraints

The steps to create security constraints for basic and custom (application container) authentication are almost the same. The difference between the two types of authentication is how you pass credentials to the server from the client application. The steps to set up security are as follows:

- 1 In the services-config.xml file, specify the login command for your application server by using the `<security>` tag. For more information, see “[Configuring security](#)” on page 432.
- 2 In the services-config.xml file, define a security constraint. A security constraint can specify basic or custom authentication, and it can define one or more roles. For more information, see “[Configuring security](#)” on page 432.
- 3 In your destination definition, reference the security constraint. For more information, see “[Configuring a destination to use a security constraint](#)” on page 434.
- 4 If you use custom authentication, pass the credentials to the server by calling the `ChannelSet.login()` method. A `result` event indicates that the login occurred successfully, and a `fault` event indicates the login failed. For more information, see “[Custom authentication](#)” on page 437. For basic authentication, you do not have to modify your Flex application. The browser opens a login dialog box when you first attempt to connect to the destination. Use that dialog box to pass credentials.

Configuring security

Typically, you configure security in the services-config.xml file. However, you can also add security settings specific to other services in the data-management-config.xml, messaging-config.xml, and other configuration files.

You reference the security constraints when you define a destination. Multiple destinations can share the same security constraints. The following example shows a security definition in the services-config.xml file that defines two security constraints. One security constraint uses basic authentication and the other uses custom (application container) authentication:

```
<services-config>
    <security>
        <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
            <per-client-authentication>false</per-client-authentication>
        </login-command>

        <security-constraint id="trusted">
            <auth-method>Basic</auth-method>
            <roles>
                <role>guests</role>
                <role>accountants</role>
                <role>employees</role>
                <role>managers</role>
            </roles>
        </security-constraint>

        <security-constraint id="sample-users">
            <auth-method>Custom</auth-method>
            <roles>
                <role>sampleusers</role>
            </roles>
        </security-constraint>
    </security>
    ...
</services-config>
```

This example sets `class` and `server` properties for the `login-command` for the Tomcat server.

The following table describes the properties that you use to configure security:

Property			Description
security			The top-level tag in a security definition. Use this tag as a child tag of the <services-config> tag.
	login-command		<p>Specifies the login command and application server. LiveCycle Data Services supplies the following login commands: TomcatLoginCommand (for both Tomcat and JBoss), JRunLoginCommand, WeblogicLoginCommand, WebSphereLoginCommand, OracleLoginCommand.</p> <p>For JBoss, set the class property to flex.messaging.security.TomcatLoginCommand, but set the server property to JBoss.</p> <p>If you set the server property to all, the login command is used regardless of your application server.</p>
		per-client-authentication	Set to true to enable per-client authentication. The default value is false. For more information, see “ Using per-client and per-session authentication ” on page 433.
	security-constraint		Define a security constraint.
		auth-method	Specifies the authentication method as either Basic or Custom. The default value is Custom.
		roles	<p>Specifies the authorization roles. The roles specified by the destination must be enabled to perform the request operation on the application server.</p> <p>Roles are application-server specific. For example, in Tomcat, they are defined in conf/tomcat-user.xml file.</p>

Using per-client and per-session authentication

By default, authentication is performed on a per-session basis and authentication information is stored in the FlexSession object associated with the Flex application. Therefore, if two client-side applications share the same session, when one is authenticated for the session, the other one is also authenticated. For example, multiple tabs in the same web browser share the same session. If you open a Flex application in one tab, and then authenticate, any copy of that application running in another tab is also authenticated.

Alternatively, you can enable per-client authentication by setting the per-client-authentication property of the login-command to true. Authentication information is then stored in the FlexClient object associated with the Flex application. Setting it to true allows multiple clients sharing the same session to have distinct authentication states. For example, two tabs of the same web browser could authenticate users independently.

Another reason to set the per-client-authentication property to true is to configure the server to reset the authentication state based on a browser refresh. Refreshing a browser does not create a new server session. Therefore, by default, the authentication state of a Flex application persists across a browser refresh. Setting perclientauthentication-- to true resets the authentication state based on the browser state.

The login commands that ship with LiveCycle Data Services rely on authentication information stored in the J2EE session on the server. If you use one of these login commands with per-client authentication, the authentication information for the different clients conflicts in the J2EE session. Therefore, to use per-client authentication, create a custom login command that does not store authentication information in the J2EE session.

Configuring a destination to use a security constraint

Define a security constraint in the security section of the Flex services configuration file and reference it in destinations, or in model entities if you are using model-driven development. The following example shows a security constraint that is referenced in two destination definitions:

```
<security>
    <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat">
        <per-client-authentication>false</per-client-authentication>
    </login-command>

    <security-constraint id="trusted">
        <auth-method>Basic</auth-method>
        <roles>
            <role>employees</role>
            <role>managers</role>
        </roles>
    </security-constraint>
</security>

<service>
    <destination id="SecurePojol">
        ...
        <security>
            <security-constraint ref="trusted"/>
        </security>
    </destination>

    <destination id="SecurePoj02">
        ...
        <security>
            <security-constraint ref="trusted"/>
        </security>
    </destination>
    ...
</service>
```

When a security constraint applies to only one destination, you can declare the security constraint in the destination definition, as shown in the following example:

```
<destination id="roDest">
    ...
    <security>
        <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
                <role>roDestUser</role>
            </roles>
        </security-constraint>
    </security>
</destination>
```

You can set a default security constraint at the service level. Any destination of a service that does not have an explicit security constraint uses the default security constraint. This is true for non-model and model-driven development. The following example shows a default security constraint configuration:

```
<service>
...
    <default-security-constraint ref="sample-users"/>
</service>
```

The following table describes the properties that you use to configure security for a destination:

Property			Description
destination			
	security		<p>Specifies a security constraint for the destination.</p> <p>Use the <code>ref</code> attribute to specify the name of a security constraint. Or, you can define a local security constraint by using the syntax described in the section “Configuring security” on page 432.</p>
	properties		
		server	<p>Specifies security constraints specific to the adapter used by the destination. Use the <code>ref</code> attribute to specify the name of a security constraint.</p> <p>For the ASObjectAdapter and JavaAdapter Data Management Service adapters , you can use the following properties to specify a security constraints for each type of data management operation:</p> <ul style="list-style-type: none">• <code>count-security-constraint</code>• <code>create-security-constraint</code>• <code>read-security-constraint</code>• <code>update-security-constraint</code>• <code>delete-security-constraint</code>• <code>send-security-constraint</code>• <code>subscribe-security-constraint</code> <p>For model-driven development, you use <code>operationname-security-constraint-ref</code> annotations on an entity.</p> <p>For the ActionScriptAdapter used with the Message Service, you can use the following properties to specify a security constraint:</p> <ul style="list-style-type: none">• <code>send-security-constraint</code>• <code>subscribe-security-constraint</code> <p>For more information, see “Using the Message Service” on page 190.</p>
		include-methods	For Remoting Service destinations, specifies the list of methods that the destination is able to call. For more information, see “ Restricting method access on a Remoting Service destination ” on page 436.

Property			Description
		exclude-methods	For Remoting Service destinations, specifies the list of methods that the destination is prohibited from calling. For more information, see “ Restricting method access on a Remoting Service destination ” on page 436.
service			
	default-security-constraint		Specifies the default security constraint for all destinations that do not explicitly define a security constraint.

Restricting method access on a Remoting Service destination

For Remoting Service destinations, you can declare destinations that only allow invocation of methods that are explicitly included in an include list. You can use this feature with or without a security constraint on the destination. Any attempt to invoke a method that is not in the `include-methods` list results in a `fault` event. For even more granular security, you can assign a security constraint to one or more methods in the `include-methods` list. If a destination-level security constraint is defined, it is tested first and method-level constraints are checked second.

The following example shows a destination that contains an `include-methods` list and a method-level security constraint on one of the methods in the list:

```
<destination id="sampleIncludeMethods">
    <properties>
        <source>my.company.SampleService</source>
        <include-methods>
            <method name="fooMethod"/>
            <method name="barMethod" security-constraint="admin-users"/>
        </include-methods>
    </properties>
    <security>
        <security-constraint ref="sample-users"/>
    </security>
</destination>
```

You can also use an `exclude-methods` element, which is like the `include-methods` element, but operates in the reverse direction. All public methods on the source class are visible to clients except for the methods in this list. Use `exclude-methods` if only a few methods on the source class must be hidden and no methods require a tighter security constraint than the destination.

Basic authentication

Basic authentication relies on standard J2EE basic authentication from the application server. When you use basic authentication to secure access to destinations, you typically secure the endpoints of the channels that the destinations use in the `web.xml` file. You then configure the destination to access the secured resource to be challenged for a user name (principal) and password (credentials).

For basic authentication, LiveCycle Data Services checks that a currently authenticated principal exists before routing any messages to the destination. If no authenticated principal exists, the server returns an HTTP 401 error message to indicate that authentication is required. In response to the HTTP 401 error message, the browser prompts the user to enter a user name and password. The web browser performs the challenge independently of the Flex client application. After the user successfully logs in, they remain logged in until the browser is closed.

Note: Basic authentication cannot be used for NIO and RTMP channels. Therefore, use custom authentication with RTMP and NIO channels.

The following example shows a security constraint definition that specifies roles for authorization:

```
<security-constraint id="privileged-users">
    <auth-method>Basic</auth-method>
    <roles>
        <role>privilegedusers</role>
        <role>admins</role>
    </roles>
</security-constraint>
```

Custom authentication

For custom authentication, the client application passes credentials to the server without relying on the browser. LiveCycle Data Services provides application container authentication with *login command* classes for supported application servers; the login command class implements an interface called `LoginCommand` and overrides `doAuthentication()`, `doAuthorization()`, and `logout()` methods to do container-specific authentication and authorization. When a FlexClient tries to log in, `LoginCommand.doAuthentication()` is called and returns a Principal that is set on the `FlexSession` object for subsequent requests. Next, as a Flex client tries to call the destination, `LoginCommand.doAuthorization` is called for each request.

Optionally, you can write a custom login command class when application container authentication does not meet your needs. For more information, see “[Login commands](#)” on page 429.

Although you apply security constraints to a destination, you actually log in and log out of the channels associated with the destination. Therefore, to send authentication credentials to a destination that uses custom authentication, you specify a user name and password as arguments to the `ChannelSet.login()` method. You remove credentials by calling the `ChannelSet.logout()` method.

The `ChannnelSet.login()` and `ChannelSet.logout()` methods are the preferred methods for setting and removing credentials. In the previous release of LiveCycle Data Services, you sent credentials to a destination by calling the `setCredentials()` method of a component such as `RemoteObject`, `Producer`, `Consumer`, `WebService`, or `HTTPService`. The `setCredentials()` method did not actually pass the credentials to the server until the first attempt by the component to connect to the server. Therefore, if the component issued a `fault` event, you could not be certain whether the fault happened because of an authentication error, or for another reason.

The `ChannelSet.login()` method connects to the server when you call it so that you can handle an authentication issue immediately. Similarly, in the previous release of LiveCycle Data Services, you removed credentials from a component, such as `RemoteObject`, `Producer`, `Consumer`, `WebService`, or `HTTPService`, with the component’s `logout()` method. However, this method only sends a logout request to the server if the client is connected and authenticated. If these conditions are not met the legacy behavior for this method is to do nothing other than clear any credentials that have been cached for use in automatic reconnects. The `ChannelSet.logout()` method removes credentials immediately.

Because multiple destinations can use the same channels, and corresponding `ChannelSet` object, logging in to one destination logs the user in to any other destination that uses the same channel or channels. If two components apply different credentials to the same `ChannelSet` object, the last credentials applied are used. If multiple components use the same authenticated `ChannelSet` object, calling the `logout()` method logs all components out of the destinations.

The `login()` and `logout()` methods return an `AsyncToken` object. Assign event handlers to the `AsyncToken` object for the `result` event to handle a successful call, and for the `fault` event to handle a failure.

How the server processes the `logout()` method depends on the setting of the `per-client-authentication` property:

- If the `per-client-authentication` property is `false` (default), the `logout()` method invalidates the current session. A new FlexSession object is automatically created to replace it, and the new session does not contain any attributes or an authenticated information. If authentication information is stored at the session level, and a client disconnect results in the FlexSession being invalidated, authenticated information is also cleared.
- If the `per-client-authentication` property is `true`, the `logout()` method clears the authentication information stored in the FlexClient object, but does not invalidate the FlexClient object or FlexSession object.

Note: Calling the `login()`, `setCredentials()`, or `setRemoteCredentials()` method has no effect when the `useProxy` property of a component is set to `false`.

Custom authentication client example

The following MXML example uses the `channelSet.login()` and `ChannelSet.logout()` methods with a `RemoteObject` control. This application performs the following actions:

- Creates a `ChannelSet` object in the `creationComplete` handler that represents the channels used by the `RemoteObject` component.
- Passes credentials to the server by calling the `ROLogin()` function in response to a `Button click` event.
- Uses the `RemoteObject` component to send a String to the server in response to a `Button click` event. The server returns the same String back to the `RemoteObject` component.
- Uses the `result` event of the `RemoteObject` component to display the String in a `TextArea` control.
- Logs out of the server by calling the `ROLogout()` function in response to a `Button click` event.

```
<?xml version="1.0"?>
<!!-- security/SecurityConstraintCustom.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="100%" height="100%" creationComplete="creationCompleteHandler();">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.messaging.config.ServerConfig;
            import mx.rpc.AsyncToken;
            import mx.rpc.AsyncResponder;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.messaging.ChannelSet;

            // Define a ChannelSet object.
            public var cs:ChannelSet;

            // Define an AsyncToken object.
            public var token:AsyncToken;

            // Initialize ChannelSet object based on the
            // destination of the RemoteObject component.
            private function creationCompleteHandler():void {
                if (cs == null)
                    cs = ServerConfig.getChannelSet(remoteObject.destination);
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```
// Login and handle authentication success or failure.
private function ROLogin():void {
    // Make sure that the user is not already logged in.
    if (cs.authenticated == false) {
        token = cs.login("sampleuser", "samplepassword");
        // Add result and fault handlers.
        token.addResponder(new AsyncResponder(LoginResultEvent, LoginFaultEvent));
    }
}

// Handle successful login.
private function LoginResultEvent(event:ResultEvent, token:Object=null):void {
    switch(event.result) {
        case "success":
            authenticatedCB.selected = true;
            break;
        default:
    }
}

// Handle login failure.
private function LoginFaultEvent(event:FaultEvent, token:Object=null):void {
    switch(event.fault.faultCode) {
        case "Client.Authentication":
            default:
                authenticatedCB.selected = false;
                Alert.show("Login failure: " + event.fault.faultString);
    }
}

// Logout and handle success or failure.
private function ROLogout():void {
    // Add result and fault handlers.
    token = cs.logout();
    token.addResponder(new AsyncResponder(LogoutResultEvent, LogoutFaultEvent));
}

// Handle successful logout.
private function LogoutResultEvent(event:ResultEvent, token:Object=null):void {
    switch (event.result) {
        case "success":
            authenticatedCB.selected = false;
            break;
        default:
    }
}

// Handle logout failure.
private function LogoutFaultEvent(event:FaultEvent, token:Object=null):void {
    Alert.show("Logout failure: " + event.fault.faultString);
}

// Handle message received by RemoteObject component.
private function resultHandler(event:ResultEvent):void {
    ta.text += "Server responded: " + event.result + "\n";
}
```

```
// Handle fault from RemoteObject component.
private function faultHandler(event:FaultEvent):void {
    ta.text += "Received fault: " + event.fault + "\n";
}
]]>
</mx:Script>

<mx:HBox>
<mx:Label text="Enter a text for the server to echo"/>
<mx:TextInput id="ti" text="Hello World!"/>
<mx:Button label="Login"
    click="ROLogin();"/>
<mx:Button label="Echo"
    enabled="{authenticatedCB.selected}"
    click="remoteObject.echo(ti.text);"/>
<mx:Button label="Logout"
    click="ROLogout();"/>
<mx:CheckBox id="authenticatedCB"
    label="Authenticated?"
    enabled="false"/>
</mx:HBox>
<mx:TextArea id="ta" width="100%" height="100%"/>

<mx:RemoteObject id="remoteObject"
    destination="remoting_AMF_SecurityConstraint_Custom"
    result="resultHandler(event);"
    fault="faultHandler(event);"/>
</mx:Application>
```

Configure Tomcat for custom authentication

Perform the following configuration steps to use custom (application container) authentication with Tomcat:

- 1 Place the flex-tomcat-common.jar file in the tomcat/lib/lcds directory.
- 2 Place the flex-tomcat-server.jar file in the tomcat/lib/lcds directory.
- 3 Edit the tomcat/conf/catalina.properties file to add the following path to the common.loader property:
 `${catalina.home}/lib/lcds/*.jar`
- 4 Edit the tomcat/conf/context.xml file to add the following tag to the Context descriptors:
`<Valve className="flex.messaging.security.TomcatValve"/>`
- 5 Restart Tomcat.

You can now authenticate against the current Tomcat realm. Typically user information is in the conf/tomcat-users.xml file. For more information, see the Tomcat documentation.

Passing credentials to HTTP services and web services

A security constraint defines the security restrictions on a LiveCycle Data Services destination. However, you can also use LiveCycle Data Services as a proxy to a remote HTTP service or web service.

If the remote HTTP service or web service requires credentials to control access, use the `setRemoteCredentials()` method to pass the credentials. For example, when using an `HTTPService` component, you can pass credentials to a secured JSP page. Passing remote credentials is distinct from passing credentials to satisfy a security constraint that you defined on the destination. However, you can use the two types of credentials in combination. The credentials are sent to the destination in message headers.

You can also call the `setRemoteCredentials()` method for Remoting Service destinations managed by an external service that requires user name and password authentication, such as a ColdFusion Component (CFC).

The following example shows ActionScript code for sending remote user name and remote password values to a destination. The destination configuration passes these credentials to the actual JSP page that requires them.

```
var employeeHTTP:mx.rpc.http.HTTPService = new HTTPService();
employeeHTTP.destination = "secureJSP";
employeeHTTP.setRemoteCredentials("myRemoteUserName", "myRemotePassword");
employeeHTTP.send({param1: 'foo'});
```

As an alternative to setting remote credentials on a client at run time, you can set remote credentials in `remote-username` and `remote-password` elements in the properties section of a server-side destination definition. The following example shows a destination that specifies these properties:

```
<destination id="samplesProxy">
    <channels>
        <channel ref="samples-amf"/>
    </channels>
    <properties>
        <url>
            http://someserver/SecureService.jsp
        </url>
        <remote-username>johndoe</remote-username>
        <remote-password>opensaysme</remote-password>
    </properties>
</destination>
```

Whitelist and blacklist filtering with NIO endpoints

You can protect HTTP-based and RTMP-based NIO endpoints by using whitelists and blacklists that list specific firewall, router, or web server IP addresses. A whitelist contains client IP addresses that are permitted to access endpoints. A blacklist contains client IP addresses that are restricted from accessing endpoints.

The blacklist takes precedence over the whitelist in the event that the client IP address is a member of both the whitelist and the blacklist.

The `whitelist` and `blacklist` elements can contain 0-N `ip-address` and `ip-address-pattern` elements. The `ip-address` element supports simple IP address matching, allowing individual bytes in the address to be designated as a wildcard by using the asterisk character (*).

The `ip-address-pattern` element supports regular expression pattern matching of IP addresses. Regular expressions allow for powerful range-based IP address filtering.

The following example shows a whitelist and a blacklist:

```
<whitelist>
  <ip-address-pattern>237.*</ip-address-pattern>
  <ip-address>10.132.64.63</ip-address>
</whitelist>

<blacklist>
  <ip-address>10.60.147.*</ip-address>
  <ip-address-pattern>10\\.\132\\.\17\\.\5 [0-9] {1,2}</ip-address-pattern>
</blacklist>
```

The IP addresses that you pass to the `ip-address` property can be in IPv4 and IPv6 formats.

By default, the server that an endpoint starts up binds to all local network interface cards (NICs) on the specified port. The port is based on the endpoint URL. You can specify a `<bind-address>xxx.xxx.xxx.xxx</bind-address>` element in the properties section of the channel definition to bind the server to a specific local NIC on startup.

You use this element when your computer has an internally facing NIC and a public NIC. You can bind the server to just the internal NIC, for example, to protect it from public access. You can also use the `<bind-port/>` property to connect to a public port on a firewall or load balancer that forwards traffic to an internal port. This property lets the channel work smoothly with hardware load balancers or proxies that require access to the server over a standard port, but when the hop from the load balancer to the server must be a non-standard port.

By default, the accept socket of the server uses a server platform default to control the size of the queue of pending connections to accept. You can use the `accept-backlog` element in the channel properties section to control the size of this queue.

For more information, see “[Channels and endpoints](#)” on page 38.

Clustering

Clusters are a group of servers that function as a single server. LiveCycle Data Services supports message and data routing across a cluster so that clients connected to different servers in the cluster can exchange information. LiveCycle Data Services also supports channel failover across the cluster. If a server in the cluster fails, a client connected to that server automatically attempts to connect to another server in the cluster.

Server clustering

LiveCycle Data Services provides a software clustering capability where you install LiveCycle Data Services on each server in the cluster. LiveCycle Data Services clustering provides two main features:

- Cluster-wide message and data routing

A MessageService or DataService client connected to one server can exchange messages and data with a client connected to another server. By distributing client connections across the cluster, one server does not become a processing bottleneck.

- Channel failover support

If the channel over which a Flex client communicates with the server fails, the client reconnects to the same destination on a different server or channel. Clustering handles failover for all channel types for RPC (remote procedure call), Data Management Service, and Message Service destinations.

LiveCycle Data Services supports both vertical and horizontal clusters (where multiple instances of LiveCycle Data Services are deployed on the same or different physical servers). When a LiveCycle Data Services instance starts, clustered destinations broadcast their availability and reachable channel endpoint URIs to all peers in the cluster.

You can also implement hardware clustering, which requires additional hardware. Hardware clustering typically uses a set of load balancers positioned in front of servers. Load balancers direct requests to individual servers and pin client connections from the same client to that server. Although hardware clustering is possible without using LiveCycle Data Services software clustering, it can also work in conjunction with software clustering.

Handling channel failover

When a client-side application initially connects through a channel to a destination on a clustered server, it receives a set of channel endpoint URIs for that channel and destination across all servers in the cluster. These endpoints are assigned to the `Channel.failoverURIs` property.

If a connection through a channel to a destination fails, the client uses the `Channel.failoverURIs` property to reconnect to the same channel and destination on a different server. The client attempts to connect to each server in the cluster until it successfully connects.

A destination can define multiple channels. If the client fails to connect to a server in the cluster using the original channel and destination, the client tries to reconnect to the original server and destination through a different channel. If that fails, the client then attempts to connect to a different server. This process repeats until the client has attempted to connect to all servers over all channels defined for the destination.

Note: *The list of URIs across the cluster for each channel is keyed by the channel id. Therefore, the destination configuration on all servers in the cluster must define the same set of channels.*

The crossdomain.xml file

A crossdomain.xml file provides a way for a server to indicate that its data and documents are available to SWF files served from specific domains, or from all domains. All servers in the cluster must have compatible crossdomain.xml files so that if one server fails and its clients are redirected to another server in the cluster, the second server can communicate with it.

For more information on the crossdomain.xml file, see the Flex documentation set.

Setting component properties for clustering

Different client-side components, such as Producer, Consumer, DataService, and the RPC components, define properties that control the action of the component when a channel connection fails. The following table describes these properties:

Component	Property	Description
Producer	reconnectAttempts reconnectInterval	<p><code>reconnectAttempts</code> specifies the number of times the component attempts to reconnect to a channel following a connection failure. If the component fails to connect after the specified number of attempts, it dispatches a <code>fault</code> event. The <code>fault</code> event does not occur until the component attempts to connect to all servers in the cluster, across all available channels. A value of -1 specifies to continue indefinitely and a value of 0 disables attempts. The default value is 0.</p> <p><code>reconnectInterval</code> specifies the interval, in milliseconds, between attempts to reconnect. Setting the value to 0 disables reconnection attempts. Set the value to a period of time long enough for the underlying protocol to complete its attempt and time out if necessary. The default is value 5000 (5 seconds).</p>
Consumer DataService	resubscribeAttempts resubscribeInterval	<p><code>resubscribeAttempts</code> specifies the number of times the component attempts to resubscribe to the server following a connection failure. If the component fails to subscribe to a server after the specified number of attempts, it dispatches a <code>fault</code> event. The <code>fault</code> event does not occur until the component attempts to connect to all servers in the cluster, across all available channels. A value of -1 specifies to continue trying to connect indefinitely and a value of 0 disables attempts. For the Consumer component, the default value is 5. For the DataService component, the default value is -1.</p> <p><code>resubscribeInterval</code> specifies the interval, in milliseconds, between attempts to resubscribe. Setting the value to 0 disables resubscription attempts. Set the value to a period of time long enough for the underlying protocol to complete its attempt and time out if necessary. The default is value 5000 (5 seconds).</p> <p>The combination of the <code>resubscribeAttempts</code> and <code>resubscribeInterval</code> properties defines the duration of reconnection attempts. However, this doesn't necessarily determine how many raw resubscribe requests are sent over the network. If the underlying network transport is slow in timing out a request to an unreachable host for whatever reason, the can be fewer than five resubscribe requests sent. For example, the component sends the first resubscribe attempt immediately following detection of a connection drop. If that initial request is still outstanding five seconds later, the component doesn't send another request .</p>
HTTPService RemoteObject WebService Producer Consumer DataService	requestTimeout	<p>The request timeout, in seconds, for sent messages. If an acknowledgment, response, or fault is not received from the remote destination before the timeout is reached, the component dispatches a <code>fault</code> event on the client. A value less than or equal to zero prevents request timeout.</p> <p>For more information, see "Cluster-wide message and data routing" on page 445.</p>
Channel services-config.xml file	connectTimeout connect-timeout-seconds	<p><code>Channel.connectTimeout</code> and the <code>connect-timeout-seconds</code> property in the <code>services-config.xml</code> file specify the connect timeout, in seconds, for the channel.</p> <p>A value of 0 or below indicates that a connect attempt never times out on the client. For channels that are configured to failover, this value is the total time to wait for a connection to be established. It is not reset for each failover URI that the channel attempts to connect to.</p>

To speed up the process when channels are failing over, or to detect a hung connection attempt and advance past it, you can set the `Channel.connectTimeout` property on your channels in ActionScript, or set the `<connect-timeout-seconds>` property in your channel configuration, as the following example shows:

```
<channel-definition id="triage-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="..." class="flex.messaging.endpoints.AMFEEndpoint"/>
    <properties>
        <serialization>
            <ignore-property-errors>true</ignore-property-errors>
            <log-property-errors>true</log-property-errors>
        </serialization>
        <connect-timeout-seconds>2</connect-timeout-seconds>
    </properties>
</channel-definition>
```

Client load balancing with a cluster

Using a load balancer in combination with RTMP connections is not recommended with applications for which you want to take advantage of reliable messaging. To improve support for client connectivity to a cluster of LiveCycle Data Services servers when you cannot use a load balancer, you can define a `client-load-balancing` subelement in the `properties` section of the endpoint's `channel-definition` in the `services-config.xml` file. Client applications that you compile against the `services-config.xml` file use this set of URLs to connect to the server rather than the `url` attribute specified in the channel definition's `endpoint` element. In this case, the `endpoint` URL value is not compiled into the client SWF file. Additionally, when runtime configuration is generated and returned, it contains the `client-load-balancing` list of URLs and not the `endpoint` URL.

Before the client initially connects, it shuffles through the full set of URLs specified in the `client-load-balancing` element. The client assigns one URL at random as the primary URL for its `Channel` object. It assigns the remaining URLs to the `failoverURIs` property on its `Channel` object. This process randomly distributes client connections across available LiveCycle Data Services instances in the absence of a load balancer. The following example channel definition from a `services-config.xml` file shows a `client-load-balancing` element:

```
<channel-definition id="rtmp" class="mx.messaging.channels.RTMPChannel">
<endpoint url="rtmp://10.132.19.65:1935" class="flex.messaging.endpoints.RTMPEndpoint"/>
    <properties>
        <client-load-balancing>
            <url>rtmp://edge1.adobe.com:1935</url>
            <url>rtmp://edge2.adobe.com:1935</url>
            <url>rtmp://edge3.adobe.com:1935</url>
        </client-load-balancing>
    </properties>
</channel-definition>
```

In this channel definition, the `url` attribute of the `endpoint` element only defines the local port on which the endpoint binds service connections. Clients compiled against this channel definition attempt to connect to URLs listed in the `client-load-balancing` configuration element in a random fashion.

Cluster-wide message and data routing

Messaging and data service destinations broadcast messages or changes to the corresponding destination on the other servers in the cluster. Therefore, `MessageService` or `DataService` components in one client application connected to one server can receive messages or changes that a different client sends or commits to a different server.

When a client application subscribes to a particular service on one server in a cluster, and the server fails, the client can direct further messages to another server in the same cluster. This feature is available for all service types defined in the services configuration file.

For the Message Service and the Data Management Service, both failover and replication of application messaging state are supported. For the Remoting Service and Proxy Service, failover only is supported. When Remoting Service and Proxy Service use the AMF or RTMP channel, the first request that fails generates a message `fault` event, which invalidates the current channel. The next time a request is sent, the client looks for another channel, which triggers a failover event followed by a successful send to the secondary server that is still running.

The `requestTimeout` property causes the client to dispatch a `fault` event if it has not received a response (either a response, acknowledgement, or fault) from the remote destination within the interval. This action is more useful with read-only calls over HTTP if the regular network request timeout is longer than desirable.

Use the `requestTimeout` property only with calls that create, update, or delete data on the server. If a request timeout occurs, query the server for its current state before you decide whether to retry the operation. Messages sent by Producer components or calls made by RPC services are not automatically queued or resent. These messages are not *idempotent*, which means they do not produce the same result no matter how many times they are performed. The application must determine whether it can safely resend a message or call the RPC service again.

Configuring clustering

To configure clustering, configure JGroups by using the `jgroups-tcp.xml` file, and configure LiveCycle Data Services by using the `services-config.xml` file.

Configuring JGroups

LiveCycle Data Services uses JGroups, an open source clustering platform to implement clustering. By default, a LiveCycle Data Services web application is not configured to support JGroups. The `jgroups.jar` file and the `jgroups-*.xml` properties files are located in the `resources/clustering` folder of the LiveCycle Data Services installation. Before you can run an application that uses JGroups, copy the `jgroups.jar` file to the `WEB-INF/lib`, and copy the `jgroups-*.xml` files to the `WEB-INF/flex` directory.

The JGroups configuration file determines how a clustered data service destination on one server broadcasts changes to corresponding destinations on other servers. JGroups supports UDP multicast or TCP multicast to a list of configured peer server addresses. Adobe recommends using the TCP option (TCP and TCPPING options in `jgroups-tcp.xml`). A unique version of this file is required for each server in the cluster. In the TCP element, use the `bind_addr` attribute to reference the current server by domain name or IP. In the TCPPING element, you reference the other peer servers.

For example, if the cluster contains four servers, the elements in the first `jgroups-tcp.xml` config file would look like the following example:

```
<TCP bind_addr="64.13.226.47" start_port="7800" loopback="false"/>
<TCPPING timeout="3000"
    initial_hosts="data1.com[7800],data2.com[7800],data3.com[7800]"
    port_range="1"
    num_initial_members="3"/>
<FD_SOCK/>
<FD timeout="10000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
<MERGE2 max_interval="10000" min_interval="2000"/>
<pbcast.NAKACK gc_lag="100" retransmit_timeout="600,1200,2400,4800"/>
<pbcast.STABLE stability_
    delay="1000" desired_avg_gossip="20000" down_thread="false"
    max_bytes="0" up_thread="false"/>
<pbcast.GMS
    print_local_addr="true" join_timeout="5000" join_retry_timeout="2000" shun="true"/>
```

- The TCP `bind_addr` and `initial_hosts` attributes vary for each server in the cluster. The `bind_addr` property must be set to the IP or DNS name depending on how the `/etc/hosts` file is set up on that computer.
- The TCPPING `initial_hosts` setting lists all other nodes in the cluster, not including the node corresponding to the configuration file. Set the `num_initial_members` to the number of other nodes in the cluster.
- The FD (Failure Detection) setting specifies to use sockets to other nodes for failure detection. This node does not consider other nodes to be out of the cluster unless the socket to the node is closed.

For information about JGroups, see <http://www.jgroups.org/javagroupsnew/docs/index.html>.

Configuring LiveCycle Data Services

Define one or more software clusters in the `clusters` section of the `services-config.xml` file. Each `cluster` element must have an `id` attribute, and a `properties` attribute that points to a JGroups properties file. The following example shows the definition of a cluster in the `services-config.xml` file:

```
<?xml version="1.0"?>
<services-config>
    ...
    <clusters>
        <cluster id="default-cluster" properties="jgroups-tcp.xml"/>
    </clusters>
    ...
</services-config>
```

The `cluster` element can take the following attributes:

Attribute	Description
<code>id</code>	Identifier of the cluster definition. Use the same <code>id</code> value across all members of a cluster.

Attribute	Description
properties	Name of a JGroups properties file.
default	A value of <code>true</code> sets a cluster definition as the default cluster. All destinations use this cluster unless otherwise specified by the cluster property in the destination definition. The <code>default</code> attribute lets you enable clustering for all destinations without having to modify your destination definitions.
url-load-balancing	The server gathers the endpoint URLs from all servers in the cluster and sends them to clients during the initial connection handshake. Therefore, clients can implement failover between servers with different domain names. For RTMP-based channels/endpoints, use the default <code>url-load-balancing</code> value, which is <code>true</code> . For HTTP-based channels/endpoints, if you use a hardware or software load balancer that supports sticky sessions between clients and servers, set the <code>url-load-balancing</code> attribute to <code>false</code> . In this mode, the client connects to each channel using the same URL and the load balancer routes requests to an available server. For this case, compile clients against a services-config.xml file with a channel endpoint URL that points at the load balancer. The load balancer pins a client to a single server, which is required to manage its subscription state efficiently. If the client's connection fails, it reconnects through the load balancer and is pinned to a new backing server where it automatically resubscribes. When <code>url-load-balancing</code> is <code>true</code> (the default), you cannot use <code>{server.name}</code> and <code>{server.port}</code> tokens in channel endpoint URLs. Instead, specify the unique server name and port for each server.

The cluster element can contain a properties element with the following optional child elements that set properties on the JGroups channel:

- channel-block
- channel-autogetstate
- channel-reconnect

These elements set the following properties on the on the JGroups channel: Channel.BLOCK, Channel.AUTO_GETSTATE, Channel.AUTO_RECONNECT, and Channel.LOCAL properties. Consult the JGroups documentation for details.

The following example shows a channel definition with a valid server name:

```
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://companyserver:8400/lcds-samples/messagebroker/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>
```

Notice that the channel endpoint omits the `{server.name}` and `{server.port}` tokens in the endpoint URL.

When you are not using model-driven development, reference the cluster in the `network` section of a destination. The value of the `ref` attribute must match the value of the `id` attribute of the cluster, as the following example shows:

```
<destination id="MyDestination">
  ...
  <properties>
    <network>
      <cluster ref="cluster1"/>
    </network>
  </properties>
  ...
</destination>
```

For model-driven development, reference a cluster in the cluster-ref annotation of an entity, as the following example shows:

```
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="cluster-ref">
      cluster1
    </item>
  </annotation>
  ...
</entity>
```

If you do not specify a cluster reference, the destination uses the default cluster definition, if one is defined. This is true for model-driven development and non-model-driven development.

Configuring cluster message routing

For publish-subscribe messaging, you have the option of using the server-to-server messaging feature to route messages sent on any publish-subscribe messaging destination. Set server-to-server messaging in a configuration property on a messaging destination definition to make each server store the subscription information for all clients in the cluster. When you enable server-to-server messaging, data messages are routed only to the servers with active subscriptions, but subscribe and unsubscribe messages are broadcast across the cluster.

Server-to-server messaging provides the same functionality as the default messaging adapter, but improves the scalability of the messaging system for many use cases in a clustered environment. In many messaging applications, the number of subscribe and unsubscribe messages is much lower than the number of data messages that are sent. To enable server-to-server messaging, you set the value of the `cluster-message-routing` property in the server section of a messaging destination to `server-to-server`. The default value is `broadcast`. The following example shows a `cluster-message-routing` property set to `server-to-server`:

```
<destination id="MyTopic">
  <properties>
    ...
    <server>
      ...
      <cluster-message-routing>server-to-server</cluster-message-routing>
    </server>
  </properties>
  ...
</destination>
```

Configuring a shared back-end system

In addition to providing client failover, a cluster sometimes processes one message on all nodes. This situation usually occurs when no back-end resource is available to coordinate the common cluster state. When no shared back-end system is available, a cluster node directs the other node to reprocess and broadcast a message. When a shared back-end system is available, a node directs the other nodes to broadcast the message to connected clients, but it does not direct the other nodes to reprocess the message.

When the `shared-backend` property is set to `true`, a cluster node directs the other nodes to broadcast messages to connected clients, but it does not direct the other nodes to reprocess the message. The `shared-backend` property is optional and is relevant only for Data Management Service destinations.

The following example shows a shared-backend property in a destination in the `data-management-config.xml` file:

```
<destination id="MyDestination">
    ...
    <properties>
        <network>
            <cluster ref="cluster1" shared-backend="true"/>
        </network>
    </properties>
    ...
</destination>
```

For model-driven development, you use a shared-backend annotation on an entity, as the following example shows:

```
<entity name=" User" persistent="true">
    <annotation name="DMS">
        <item name="cluster-ref">
            cluster1
        </item>
        <item name=
            "shared-backend">true
        </item>
    </annotation>
</entity>
```

Using the `shared-backend="false"` configuration attribute does not guarantee a single consistent view of the data across the cluster. When two clients are connected to different servers in the cluster, if they perform conflicting updates to the same data value, each server applies these changes in a different order. The result is that clients connected to one server see one view of the data, while clients connected to other servers see a different view. The conflict detection mechanism does not detect these changes as conflicts. If you require a single consistent view of data where two clients are updating the same data values at roughly the same time, use a database or other mechanism to ensure that the proper locking is performed when updating data in the back-end system.

Viewing cluster information in the server-side log

The server-side logging mechanism captures log messages from the LiveCycle Data Services server. You can set the `pattern` attribute in the `services-config.xml` file to the wildcard character (*) to get all logging messages, including clustering. Alternatively, you can use the `pattern` element to filter messages. The following example filters log messages to write out only clustering, message, and endpoint messages:

```
<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Debug">
        <properties>
            <prefix>[LCDS]</prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Service.Cluster</pattern><pattern>Message.*</pattern>
            <pattern>Endpoint.*</pattern>
        </filters>
    </target>
</logging>
```

Enabling the Message.* logging pattern lets you see clients send a CLUSTER_REQUEST_OPERATION (a command message with operation 7) to the server right after their initial connect ping. The response contains the endpoint URLs for other server nodes in the cluster. These values are automatically assigned to the failoverURIs property of the channels in the client application's channel set. After that point, if the client's connection drops, its automatic reconnect/resubscribe attempts fail over through these options in addition to retrying the primary server's endpoint.

For more information on logging, see “[Logging](#)” on page 420.

Integrating Flex applications with portal servers

Using Adobe LiveCycle Data Services, you can configure Adobe Flex client applications as local portlets hosted on JBoss Portal, BEA WebLogic Portal, or IBM WebSphere Portal.

Note: The portal server feature is not available in BlazeDS.

Supported portal servers implement the portlet specification: Java Specification Request (JSR) 168. You can also enable Flex client applications for consumption by a portlet consumer by using portal servers that support Web Services for Remote Portlets (WSRP). WSRP is an OASIS standard that defines a web service interface for accessing and interacting with interactive presentation-oriented web services.

The files required to use the portal servers are located in the *installation_dir/resources/wsrf* directory where LiveCycle Data Services is installed.

Using a Flex application on a portal server

You can configure Flex client applications as local portlets on a portal server that implements the Java portlet specification (JSR 168). You can also enable Flex client applications for consumption by a portlet consumer by using portal servers that support WSRP.

A portal server facilitates the development, deployment, operation, and presentation of aggregated content and applications that can be personalized for specific users. The Java portlet specification provides a standard way to develop user-facing web application components (portlets) for portal servers. The specification defines a common API and infrastructure that allows for product-agnostic portlets. The specification defines a portlet container contract, window states and portlet modes, management of portlet preferences, user information storage, packaging and deployment, security, and portlet-specific JSP tags. For more information about the portlet specification, see <http://jcp.org/en/jsr/detail?id=168>.

WSRP uses web services to provide a standard way for compliant portals to consume portlets hosted on other portals without any programming effort. A portal that hosts a portlet is called a producer portal. A portal that consumes a portlet using WSRP is called a consumer portal. For more information about WSRP, see the OASIS WSRP page at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp.

Enabling a Flex application as a portlet

You enable a Flex application as a portlet on a local portal server by creating a portlet definition that uses the portlet class, flex.portal.GenericFlexPortlet. The GenericFlexPortlet class handles all WSRP requests and returns an HTML-snippet wrapped SWF file for the requested portlet. When using a portal that supports WSRP, the WSRP implementation sends the snippet back in a WSRP supported SOAP message. LiveCycle Data Services also provides three default JSP pages, one for each portlet view mode: view, edit, and help. The GenericFlexPortlet class uses these JSP pages by default to satisfy requests for corresponding view modes of a portlet. You can customize the look of each view mode by specifying a custom JSP page for any of the view modes.

Each portlet-name value in a portlet definition must be unique. The code in the following example portlet.xml portal deployment descriptor file makes a Flex application available as a local portlet:

```
<portlet>
    <portlet-name>CRMPortlet</portlet-name>
    <portlet-class>flex.portal.GenericFlexPortlet</portlet-class>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>VIEW</portlet-mode>
        <portlet-mode>EDIT</portlet-mode>
    </supports>
    <portlet-info>
        <title>CRM Portlet</title>
    </portlet-info>
    <portlet-preferences>
        <preference>
            <name>full_app_uri</name>
            <value>/samples/dataservice/crm/mini.swf
            </value>
            <read-only>true</read-only>
        </preference>
    </portlet-preferences>
</portlet>
```

To expose the portlet for WSRP consumption, add the following preference element to the portlet-preferences section of the portlet.xml file. This setting enables requests from SWF files to be proxied through the consumer server in accordance with the WSRP specification.

```
<preference>
    <name>channel_uri</name>
    <value>/messagebroker/amfpolling</value>
    <read-only>true</read-only>
</preference>
```

To expose the portlet for WSRP consumption in portal servers other than WebLogic Portal, you usually include a remoteable element in the portlet definition in the portlet-server-specific configuration file and set its value to true, as the following example shows:

```
<portlet-app>
  <portlet>
    <remotable>true</remotable>
    <portlet-name>CRMPortlet</portlet-name>
  </portlet>
</portlet-app>
```

The `remotable` element enables the Flex application for WSRP consumption by any consumer portal server that has specified the WSDL of your server in its WSRP configuration file.

You must set the `full_app_uri` preference in the `portlet.xml` file. Additionally, to proxy requests through the consumer server in accordance with the WSRP specification, you must set the `channel_uri` preference in the `portlet.xml` file. When the `channel_uri` value is present, the `WSRP_ENCODED_CHANNEL` variable containing a correctly formatted channel URL, is passed to the Flex application via the `FlashVars` parameter in the HTML that the `portlet-view.jsp` file generates. All traffic from the portlet's Flex application is proxied through the consumer server. Note that this increases the HTTP load on this server if you use a polling or streaming channel.

The following preferences are supported by all Flex application portlets:

Preferences	Description
<code>full_app_uri</code>	(Required) Maps this portlet to a Flex application. Can be a relative URL or a fully qualified location.
<code>channel_uri</code>	(Optional) Relative path (starting after the context root) to the channel between the Flex application and the LiveCycle Data Services server. This should be the path of the first channel used by the destinations. Only a single channel is supported for producer/consumer portlets, so failover is not supported. If you do not set the <code>channel_uri</code> preference, a Flex portlet can still be consumed directly from the producer server, but it will not be proxied through the consumer server in accordance with the WSRP specification.
<code>view_jsp</code>	Specifies the JSP file to be used when returning markup for the VIEW view mode. Default value is <code>/wsrp_jsp/portlet-view.jsp</code> .
<code>edit_jsp</code>	Specifies the JSP file to be used when returning markup for the EDIT view mode. Default value is <code>/wsrp_jsp/portlet-edit.jsp</code> .
<code>help_jsp</code>	Specifies the JSP file to be used when returning markup for the HELP view mode. Default value is <code>/wsrp_jsp/portlet-help.jsp</code> .
<code>normal_width</code>	Specifies the width of the embedded SWF file when the portlet is in the NORMAL window state. Default value is 300.
<code>normal_height</code>	Specifies the height of the embedded SWF file when the portlet is in the NORMAL window state. Default value is 300.
<code>max_width</code>	Specifies the width of the embedded SWF file when the portlet is in the MAXIMIZED window state. Default value is 500.
<code>markup_cache_secs</code>	Specifies the time in seconds during which the consumer caches the markup for this particular portlet. The default behavior is to do no caching of the HTML markup.

In addition to these preferences, you can define custom preferences. Custom preferences are ignored by the default JSP files, but can be used in custom JSP pages. Custom preferences must include a `cust` prefix.

If you copy the `wsrp-jsp` directory, which you copy from the `installation_dir/resources/wsrp` directory of the LiveCycle Data Services installation, to a location other than directly below the context root of your web application, you must set the optional `wsrp_folder` initialization parameter to that location. If you set the `wsrp_folder` initialization parameter, you also must set initialization parameters for each of the JSP pages. The following example shows the `wsrp_folder` initialization parameter and an initialization parameter for one of the JSP pages, the `portlet-view.jsp` page:

```
<portlet>
    <portlet-name>CRMPortlet</portlet-name>
    <portlet-class>flex.portal.GenericFlexPortlet</portlet-class>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>VIEW</portlet-mode>
        <portlet-mode>EDIT</portlet-mode>
    </supports>
    <init-param>
        <name>wsrp_folder</name>
        <value>/flex/support</value>
    </init-param>
    <init-param>
        <name>view_jsp</name>
        <value>/flex/support/wsrp-jsp/portlet-view.jsp</value>
    ...
</portlet>
```

View modes

You can view a portlet in one of three modes, VIEW, EDIT, and HELP. By default, the GenericFlexPortlet uses a specific template JSP page corresponding to each of these modes.

Default view modes

The default VIEW mode JSP page simply wraps the reference to the specified SWF file in an `<object>` tag, which sets the width and height to the values specified in the configuration file for this portlet. The default EDIT mode JSP page lets users modify any of the non-read-only preferences and view the values of the read-only preferences. The default HELP mode JSP page displays basic help text. An administrator can specify custom JSP pages for any view mode by setting configuration preferences in the portlet metadata file.

If the producer and consumer servers do not both support the Portlet Management interface, all preferences appear as read-only. For a remote portlet being consumed through WSRP to have modifiable preferences, the consumer must be able to clone the producer portlet. To do so, the implementations of WSRP that the consumer and producer use must implement the optional Portlet Management interface. If one of the implementations does not, all preferences of a remote portlet appear as read-only.

Custom view modes

Portal servers can also support custom view modes. You enable specific portlets for these custom view modes by setting configuration parameters in the portlet deployment descriptor (portlet.xml) file. Because the set of supported custom view modes for a particular installation is not known ahead of time, the GenericFlexPortlet class and default JSP pages do not have any logic for custom view modes. To customize wrappers based on custom view modes, you use custom JSP pages. For details on custom view modes, see the JSR 286 portal specification at <http://jcp.org/en/jsr/detail?id=286>.

Window states

An incoming markup request from a portlet consumer contains information that indicates whether the portlet in context is MAXIMIZED, MINIMIZED, or NORMAL. This lets the portlet server adjust the markup based on the size of the portlet window. When the GenericFlexPortlet processes a markup request for a portlet that is in the MINIMIZED state, it does not return a SWF file; there is no reason to cause extra network traffic if the user cannot interact with the Flex application. In this case, the GenericFlexPortlet returns text notifying the user that the Flex application is not available in MINIMIZED state. When configuring the portlet for a Flex application, the administrator can specify the width and height of the embedded SWF by using the portlet preferences previously described.

You might require an application to appear differently between MAXIMIZED and NORMAL window states. For example, it might make more sense for a particular panel or widget to be hidden in the NORMAL state and displayed in the MAXIMIZED state. To support such customization, a PORTLET_WS flashvar, which contains the current window state, is returned in SWF file URLs. You can access the value of this variable in your MXML or ActionScript code as Application.application.parameters.PORTLET_WS. Its value corresponds to one of the window states defined in the javax.portlet.WindowState class: normal, maximized, or minimized.

Adaptability to a window state requires that you design a Flex application with portlets in mind, and you use the PORTLET_WS variable in Flex application logic.

EOLAS support

From a February 2006 upgrade to Microsoft Internet Explorer forward, SWF files are not immediately interactive in Internet Explorer unless you specify them through the object or the <embed> tag by using JavaScript in a file external to the main file. If an object tag specifies a SWF file inline in an HTML file, the user sees a Click To Activate message when hovering over the Flex application. To avoid this, the default view JSP page uses a JavaScript library to create the wrappers around Flex applications. The required JavaScript library, AC_OETags.js, is located in the same directory as the three default JSP pages. If you use your own JSP pages outside of the default directory, you must copy the JavaScript library to the correct directory.

Flex Ajax Bridge and Ajax Client Library for LiveCycle Data Services

You can use the Flex Ajax Bridge and the Ajax client library for LiveCycle Data Services in the portlet-generated JSP pages as long as the necessary parts of FDMSLib.js, FABridge.js and FDMSBridge.as are included in the JSP page. Other non-Flex-based portlets running in the same portal page can communicate with Flex portlets by using the Flex Ajax Bridge and subscribing to the same destinations as the Flex portlets.

Because JavaScript libraries are loaded from the producer server in separate requests from the portlet markup, the <script src=> tags must specify the full URL of the library on the producer server and use the renderResponse.encodeURL() function to convert the URL to proxy through a consumer server if necessary. See the wsrp-jsp/portlet-view.jsp file for examples of using the portlet encodeURL() API.

Note: There is a general issue with using JavaScript with WSRP. To avoid object name collisions, the consumer server modifies the names of all HTML objects in the markup that the producer server returns. The same modification does not occur for JavaScript libraries loaded from the producer server. Thus any references from JavaScript libraries to HTML objects in the markup most likely fail on the portal page hosted by the consumer server unless you use URL rewriting between the producer and consumer servers. In this case, you can use URL rewriting to force the consumer to go through the loaded JavaScript library and modify HTML object name references just as it did for the markup itself. To use URL rewriting, follow the guidelines in section 10 of the WSRP specification.

Logging

The GenericFlexPortlet writes log messages under the category WSRP.General.

Caching of producer portlets

When the producer server returns markup to the consumer for a particular portlet, it can specify a cache markup time in seconds. If the consumer server has to get the portlet markup for an identical portal page URL during this time, it returns the markup from cache and does not communicate with the producer. The markup_cache_secs preference lets you customize the time. For this property to be meaningful, you must configure the consumer and producer servers to support markup caching in general. If you do not specify the markup_cache_secs preference, there is no markup caching.

Deploying on a portal server

You can deploy your Flex applications on portal servers that have a full LiveCycle Data Services web application installed, and on portal servers that do not have a LiveCycle Data Services web application but host a set of precompiled SWF files.

Deploying with a LiveCycle Data Services web application

When your portal contains a Flex application that is in a LiveCycle Data Services web application, you can use standard LiveCycle Data Services debug and error logging based on the debug level that you set in the services-config.xml file.

- 1 Copy flex-portal.jar from the *installation_dir/resources/wsdp/lib* directory to the WEB-INF\lib directory of your web application.
- 2 Copy the wsdp-jsp directory from the *installation_dir/resources/wsdp* directory to the root of your web application.
- 3 Follow the portal-server-specific steps that follow to create portlets for your Flex applications.

Deploying without a LiveCycle Data Services web application

You can create a portlet that uses a Flex application that is not part of a LiveCycle Data Services web application. However, you cannot use standard LiveCycle Data Services logging because that functionality is inside the LiveCycle Data Services web application.

- 1 Copy flex-portal.jar and flex-messaging-common.jar from *installation_dir/resources/wsdp/lib* directory to the WEB-INF/lib directory of your web application.
- 2 Copy the wsdp-jsp directory from *installation_dir/resources/wsdp* directory to the root of your web application.
- 3 Follow the portal server specific steps that follow to create portlets for your Flex applications.

Deploying on JBoss Portal

When deploying on JBoss Portal, consult the relevant JBoss documentation, including the following:

- General JBoss WSRP information
- JBoss administration portlet information

Configure a JBoss Portal producer server

- 1 To create a local portlet, modify the portlet.xml file in the WEB-INF directory of your web application based on the information in “[Using a Flex application on a portal server](#)” on page 451.
- 2 To create a local instance of the new portlet, either modify portlet-instances.xml in the WEB-INF directory of your web application, or use the administration portlet at <http://server:port/admin>.
- 3 To enable your portlet for WSRP consumption, update jboss-portlet.xml in the WEB-INF directory of your web application (setting the remoteable flag to true), or use the administration portlet <http://server:port/admin>, and select the Remoteable option for your portlet.

The WSDL for your producer server should be <http://server:port/portal-wsdp/MarkupService?wsdl>.

Depending on how your DNS is set up, it has a server name that is accessible only from within your local network (server name is computer name) or a server name that is accessible from outside the local network (server name is fully qualified or is an IP address). If the SERVER name in the generated WSDL does not appear correctly, you can adjust your DNS settings or manually modify the .wsdl file, for example:

/server/default/data/wsdl/portal-wsdp.war/wsdp_services.wsdl

Configure a JBoss Portal consumer server

- 1 Add the WSDL of the producer server (does not have to be a JBoss server) to jboss-portal.sar/portal-wsrp.sar/default-wsrp.xml.
- 2 Use the administration portlet at <http://server:port/admin> to get a list of available remote portlets.
- 3 Create instances of any remote portlets that you are interested in.

Deploying on WebLogic Portal

When deploying on WebLogic Portal, consult the relevant WebLogic documentation at <http://www.oracle.com/technology/documentation>.

Configure a producer server

Create a local portlet by using the WebLogic Workspace Studio (or WebLogic Workshop in earlier versions) and completing the following steps:

- 1 Create a Portal Web Project.
- 2 Create a portlet (New > Portlet).
- 3 After naming the portlet, select Java Portlet as the portlet type.
- 4 Ensure that you set Class Name to flex.portal.GenericFlexPortlet.
- 5 Modify WEB-INF/portlet.xml based on the information in “[Enabling a Flex application as a portlet](#)” on page 452. Make sure you add the `full_app_uri` preference.
- 6 Ensure that the portlet-name attribute in the portlet.xml file matches the `definitionLabel` property in the .portlet file.
- 7 Create a .portlet file for your portlet.

Considerations for producer servers

Consider the following when configuring a WebLogic Portal producer server:

- The URL of the WebLogic portal administration user interface is <http://server:port/context-root/Admin/portal.portal>.
- WebLogic supports quite a few different portlet types, as described in the *WebLogic Portlet Development Guide*. Look specifically at "Java Portlets (JSR 168)."
- To use the EDIT mode of a portlet, your users must have sufficient permission access to the PreferencePersistenceManager EJB.
- WebLogic portlets by default are enabled as WSRP producers. To disable a portlet from being WSRP consumable, set its Offer as Remote property to false in the WebLogic Workspace Studio.
- The producer server documentation describes the steps necessary to enable a project as a WSRP producer in the WebLogic Workspace IDE. The WSDL generated for your producer server should be <http://server:port/context-root/producer?wsdl>.

Depending on your DNS setup, the WSDL either has a server name that is accessible only from within your local network (server name is computer name) or a server name that is accessible from outside the local network (server name is fully qualified or is an IP address). If the SERVER name in the generated WSDL does not appear correctly, you can adjust your DNS settings or manually modify the `wsrp-wsdl-full.wsdl` file that is generated for your application.

After setting up your portlet on the producer server, you can use the Portal Management section of the portal administration user interface (<http://server:port/context-root/Admin/portal.portal>) or WebLogic Workspace Studio to include your Flex application portlet in portal pages.

Configure a consumer server

Complete the following steps to add the WSDL of the remote producer server to a WebLogic consumer server. The producer server does not have to be a WebLogic server.

- 1 Go to portal administration application `http://server:port/context-root/Admin/portal.portal`.
- 2 Go to Portal Management.
- 3 Browse to Library > Remote Producers.
- 4 Create an entry for the remote producer.

Your remote portlet appears under the Portlets tab. You can add remote portlets to pages by navigating to the page under the Portals tab.

Deploying on WebSphere Portal 5.1 or WAS 6.0

When deploying a portlet on WebSphere Portal 5.1 or WAS 6.0, consult the relevant WebSphere Portal documentation.

Consider the following when deploying on WebSphere Portal:

- The default URL of the WebSphere Application Server is `http://SERVER:9080`.
- The default URL of the WebSphere Portal administration application is `http://SERVER:9081/wps/portal`.
- The default URL of the WebSphere Application Server administration application is `http://SERVER:9060/ibm/console/`.
- Unlike JBoss Portal and WebLogic Portal, WebSphere Portal is a completely separate server with a different port and context root than the application server.

Configure a producer server

- 1 Deploy the WAR file that contains your application logic and portlet deployment configuration (`portlet.xml`) on the application server using the application server administration application. You should be able to test the non-portlet functionality of your application by starting the application server.
- 2 Add the same WAR file to the portal server's module list:
 - a Log in to the portal server administration application with `admin/admin` as the username/password.
 - b Select the Administration tab in the upper-right corner.
 - c Select the Portlet Management tab from the list on the left.
 - d Select Web Modules, and install your WAR file. This adds every portlet configured in your `portlet.xml` file to the list of portlets available from the portal server.
 - e Verify that your portlets appear under the Portlets tab on the left.

Now you can create a local portal page with your new portlets.

To make a portlet available as a WSRP producer, click the star icon next to its name under the Portlets tab. A check mark appears in the Provided column for that portlet.

The WSDL of your producer should be `http://SERVER:PRODUCER_PORT/wps/wsdl/wsrp_service.wsdl`

Configure a consumer server

Complete the following steps to add the WSDL of the remote producer server to the consumer server. The producer server does not have to be a WebSphere server.

- 1 Select Portlet Management tab on the left side under the Administration tab.

- 2** Select Web Services.
- 3** Click the New Producer button to add the producer WSDL.
- 4** Under the Web Modules tab, look for an entry with the name WSRP Producer (name you gave to remote producer) Application.
- 5** Navigate to the portlets listed under this application
- 6** Make copies of any portlets to consume.

Your remote portlets appear under the Portlets tab, and you can add them to your pages.

Chapter 12: Additional programming topics

The Ajax client library

The Ajax client library for Adobe LiveCycle Data Services is a set of JavaScript APIs that lets Ajax developers access the messaging and data management capabilities of LiveCycle Data Services directly from JavaScript. It lets you use Flex clients and Ajax clients that share data in the same messaging application or distributed data application.

About the Ajax client library

The Ajax client library for LiveCycle Data Services is a JavaScript library that lets Ajax developers access the messaging and data management capabilities of LiveCycle Data Services directly from JavaScript. Using the Ajax client library, you can build sophisticated applications that more deeply integrate with back-end data and services. LiveCycle Data Services provides publish-subscribe messaging, real-time data streaming, and a rich data management API. The Ajax client library lets you use Flex clients and Ajax clients that share data in the same messaging application or distributed data application.

When to use the Ajax client library

The Ajax client library is useful for enhancing any Ajax application with the capabilities of the Data Management Service or message service. Integration with LiveCycle Data Services can provide data push and access to data sources outside of strictly XML over HTTP. Because Ajax provides better affordance for HTML-based applications that are not strictly read-only, many Ajax applications are facilitating round tripping of data. Including the Data Management Service with these applications provides the benefits of pushed updates, conflict management, lazy loading, and direct integration with back-end domain models.

The Ajax client library does not support the Flex RPC service capabilities, which include remote objects, HTTP services, and web services.

Requirements for using the Ajax client library

To use the Ajax client library, you must have the following:

- The Ajax client library, which is included in the following directory of the LiveCycle Data Services installation:
install_root/resources/fds-ajax-bridge
- A supported Java application server or servlet container
- Adobe Flex Software Development Kit (SDK) with LiveCycle Data Services SWC files added to it; for more information, see “[Building your client-side application](#)” on page 13.
- Adobe Flash Player 9 or above
- Microsoft Internet Explorer, Mozilla Firefox, or Opera with JavaScript enabled

Using the Ajax client library

Compiling an Ajax client library SWF file

You must have the following ActionScript libraries to compile the client SWF file required for the Ajax client library:

FABridge.as The Flex Ajax Bridge library. Import FABridge.as into your Ajax client library ActionScript file; for example, the FDMSBridge.as file imports the FABridge.as file.

FDMSBase.as The base class of the Ajax client library.

FDMSBridge.as The Ajax client library.

Use the following command line (on a single line) to compile the FDMSBridge.swf file:

```
mxmclc
    -verbose-stacktraces
    -services <path_to_services-config.xml>\services-config.xml
    -context-root <web application context root>
    -o <path_to_store_compiled_swf>\FDMSBridge.swf
    <path_to_FDMSBridge.as>\FDMSBridge.as
```

This command line assumes that you added mxmclc to the system path, or you run the command from the bin directory of the Flex SDK into which you have added the LiveCycle Data Services SWC files. The FDMSBase.as and FDMSBridge.as files are located in the *install_root/resources/fds-ajax-bridge/actionscript* directory. The FABridge.as file is located in the *install_root/resources/fds-ajax-bridge/actionscript/bridge* directory.

Initializing the Ajax client library

To use the Ajax client library in an HTML file, you must include the following JavaScript libraries in script elements in the HTML file:

FDMSLib.js This file contains the definition of LiveCycle Data Services APIs as JavaScript, and some utility methods for constructing and initializing Data Management Service objects. Include this file in any HTML file that requires LiveCycle Data Services. This file requires the FABridge.js file.

FABridge.js This file provides the Flex Ajax Bridge library, which is the JavaScript gateway to Flash Player.

Typically, Flash Player is hidden on the page, because the browser performs all the rendering. The JavaScript code in the following HTML code inserts the hidden Flash Player and initializes the library:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
!-- Import the FABridge.js and FDMSLib.js scripts. -->
<script type="text/javascript" src="FABridge.js"></script>
<script type="text/javascript" src="FDMSLib.js"></script>
</head>
<body>

<script language="JavaScript" type="text/javascript">
var productService;
var products;
/*
Load the Ajax client library and call the function listed as the second
parameter, which in this case is fdmsLibraryReady().
*/
FDMSLibrary.load("FDMSBridge.swf", fdmsLibraryReady);

/*
* Retrieve data after FDMSLibrary.load has been called and the Ajax client
* library is ready.
*/
function fdmsLibraryReady()
{
productService = new DataService("inventory");
products = new ArrayCollection();
var token = productService.fill(products);
token.addResponder(new AsyncResponder(productsResult, productsFault));
token.set("operation", "fill");
}
/**
*Scatter the product data into the appropriate elements.
*/
function productsResult(event)
{
...
}

function productsFault(event)
{
document.write("Request failed.");
}

</script>
</body>
</html>
```

The `FDMSLibrary.load()` convenience method inserts HTML code that puts a hidden Flash Player on the page. This Flash Player instance uses the specified location (for example, `ajax/products/FDMSBridge.swf`) to load a compiled SWF file. You can specify any SWF file that includes the FABridge and LiveCycle Data Services API classes. Using the `FDMSBridge.as` file provides the smallest SWF file. The second argument to the `FDMSLibrary.load()` method specifies the name of the JavaScript function to call when the library is loaded. You must define the JavaScript function that you specify.

To provide a visible SWF file, you must place the appropriate embed tags in the HTML file, and you must set a `bridgeNameflashvars`, as the following example shows.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
<title>Products</title>
<link href="css/accordion.css" rel="stylesheet" type="text/css"/>
<link href="../css/screen.css" rel="stylesheet" type="text/css"/>
<script type="text/javascript" src="include/FABridge.js"></script>
<script type="text/javascript" src="include/FDMSLib.js"></script>
</head>
<body>
<script>
    document.write("<object id='flexApp'
classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' \
codebase=
    'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab'
height='400' width='400'>"); 
    document.write("<param name='flashvars' value='bridgeName=example' />"); 
    document.write("<param name='src' value='app.swf' />"); 
    document.write("<embed name='flexApp'
pluginspage='http://get.adobe.com/flashplayer' \
src='app.swf' height='400' width='400'
flashvars='bridgeName=example' />"); 
    document.write("</object>"); 
    FABridge.addInitializationCallback("example", exampleBridgeReady);
</script>
<noscript><h1>This page requires JavaScript. Please enable JavaScript
in your browser and reload this page.</h1>
</noscript>
<div id="wrap">
<div id="header">
    <h1>Products</h1>
</div>
<div id="content">
    <table id="products">
<caption>Adobe Software</caption>
<tr>
...
</body>
<script>
    function exampleBridgeReady()
    {
        var exampleBridge = FABridge["example"];
        // Do something with this specific bridge.
    }
</script>
...
</html>
```

Reading data with the Ajax client library

After the Ajax client library is loaded and initialized, it sends a call back to the specified method indicating the ready state. Then, the client constructs the appropriate Data Service objects and loads data from the server.

The following example shows this sequence. This example uses the `inventory` destination that is configured in the `data-management-config.xml` file of the `lcds-samples` web application. If compile an `FABridge.swf` file as described in “[Compiling an Ajax client library SWF file](#)” on page 461, you can run this application. The application assumes the `FABridge.js` and `FDMSLib.js` files as well as this HTML file are in the root directory of the `lcds-samples` web application. Start your application server and the samples database before using the application.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
<title>Products</title>

<!-- Import the FABridge.js and FDMSLib.js scripts. -->
<script type="text/javascript" src="FABridge.js"></script>
<script type="text/javascript" src="FDMSLib.js"></script>
</head>
<body>
<div id="wrap">
<div id="header">
<h1>Products retrieved with DataService.fill() method</h1>
</div>
<div id="content">
<table id="productTable">

<script language="JavaScript" type="text/javascript">
    var productService;
    var products;
    /*
    Load the Ajax client library and call the function listed as the second
    parameter, which in this case is fdmsLibraryReady().
    */
    FDMSLibrary.load("FDMSBridge.swf", fdmsLibraryReady);

    /*
    * Retrieve data after FDMSLibrary.load has been called and the Ajax client
    * library is ready.
    */
    function fdmsLibraryReady()
    {
        productService = new DataService("inventory");
        products = new ArrayCollection();
        var token = productService.fill(products);
        token.addResponder(new AsyncResponder(productsResult, productsFault));
        token.set("operation", "fill");
    }
    /**
     *Scatter the product data into the appropriate elements.
     */
    function productsResult(event)
    {
```

```
if (event.getToken().get("operation") == "fill")
{
    var htmlText = "<table border='1' width='500'>";
    htmlText += getHeaderRow(htmlText);

    var product;
    for (var i=0; i < products.getLength(); i++)
    {
        product = products.getItemAt(i);
        htmlText +=
            "<tr valign='top'><td>" + product.getName() +
            "</td><td>" + product.getDescription() +
            "</td><td>" + product.getPrice() +
            "</td></tr>";
    }
    htmlText += "</table>";

    // document.write(htmlText);

    document.all["productTable"].innerHTML=htmlText;
}
}

function productsFault(event)
{
    document.write("Request failed.");
}

function getHeaderRow(htmlText) {
    htmlText +=
        "<tr align='left'><th>Product" +
        "</th><th>Description" +
        "</th><th>Price" +
        "</th></tr>";
    return htmlText;
}

</script>
</body>
</html>
```

The `fdmsLibraryReady()` method loads the data the same way as in a regular Flex application. The major difference when using the Ajax client library is that you must manage the code to display data. The items returned in the specified ArrayCollection are anonymous objects on both sides of the bridge. The way in which the shim SWF file is compiled by default does not provide ActionScript 3.0 classes for the transfer objects. To support lazily loaded properties, each transfer object must call across the bridge to fetch data.

Updating data with the Ajax client library

Updating the values returned from a Data Management Service fill operation requires that the objects returned from the `fill()` operation provide getter and setter methods for each property on an anonymous object. The default serialization of the properties has the form `setXXX()` and `getXXX()`, where the `XXX` is the property name. If you don't use a framework like Spry, updating data in the application requires logic to gather data points from input controls and set those values individually.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<!-- Import the FABridge.js and FDMSLib.js scripts. -->
<script type="text/javascript" src="FABridge.js "></script>
<script type="text/javascript" src="FDMSLib.js"></script>
</head>
<body>
<script language="javascript">
    var productService;
    var products;
    var currentProductIndex;
    // index into products collection for the item currently being edited
    ...
    /*Load the Ajax client library and call the function listed as the second
    parameter, which in this case is fdmsLibraryReady().
    */
    FDMSLibrary.load("FDMSBridge.swf", fdmsLibraryReady);

    function fdmsLibraryReady()
    {
        productService = new DataService("inventory");
        products = new ArrayCollection();
        var token = productService.fill(products);
        token.addResponder(new AsyncResponder(productsResult, productsFault));
        token.set("operation", "fill");
    }
    /**
     *Scatter the product data into the appropriate elements.
     */
    function productsResult(event)
    {
        ...
    }

    function productsFault(event)
    {
        document.write("Request failed.");
    }

    /*
     *Gather the data and update the appropriate transfer object
     */
    function commitData()
    {
        var item = products.getItemAt(currentProductIndex);
        var itemData = document.all;
        item.setProductName(itemData["productName"].value);
        item.setProductCategory(itemData["productCategory"].value);
        ...
        productService.commit();
    }
    ...
</script>
</body>
</html>
```

If a property is lazily loaded and is not local, the `getXXX()` method returns null. When the item becomes available, the transfer object dispatches a `PropertyChangeEvent` event. The `ArrayCollection` also dispatches an event, but it is a `CollectionChangeEvent` event.

FDMSLibrary methods

The following list describes important methods you call directly on the `FDMSLibrary` object:

- The `FDMSLibrary.load("path", callback)` method inserts HTML code that puts a hidden Flash Player on the page. This Flash Player instance uses the specified location (for example, `ajax/products/FDMSBridge.swf`) to load a compiled SWF file. It calls the user-defined function specified in the second argument.
- The `FDMSLibrary.addRef(obj)` method increments the ActionScript reference count for the object passed as an argument. This is a wrapper function that calls the `FABridge.addRef()` method. You need to do this in order to keep references to objects (for example, events) valid for JavaScript after the scope of their dispatch function has ended.
- The `FDMSLibrary.destroyObject(object)` method directly destroys an ActionScript object by invalidating its reference across the bridge. After this method is called, any subsequent calls to methods or properties tied to the object fail.
- The `FDMSLibrary.release(obj)` method decrements the ActionScript reference count for the object passed as an argument. This is a wrapper function that in turn calls the `FABridge.release()` method. You call this method to decrease the reference count. If it gets to 0, the garbage collector cleans it up at the next pass after the change.

Limitations of the Ajax client library

The Ajax client library depends on the `FABridge` library. Additionally, two the Ajax client library JavaScript methods do not return the same thing as their ActionScript counterparts. These methods are the `ArrayCollection.refresh()` and `ArrayCollection.removeItemAt()` methods. These return an undefined type instead of a Boolean and an object.

Ajax client library API reference

The Ajax client library is a set of JavaScript APIs that lets Ajax developers access the messaging and data management capabilities of LiveCycle Data Services directly from JavaScript.

DataService

JavaScript proxy to the ActionScript `DataService` class. This class has the same API as the ActionScript 3.0 version. Each method uses a generic bridging technique to make calls and retrieve data. The exception to this rule is found in the constructor, which requires some custom bridging code; constructors in ActionScript 3.0 cannot be called by using a reflection technique and passing arbitrary parameters.

Example

```
...
<script>
...
function fdmsLibraryReady()
{
    productService = new DataService("Products");
    products = new ArrayCollection();
    var token = productService.fill(products);
    token.addResponder(new AsyncResponder(productsResult, productsFault));
}
...
</script>
```

DataStore

JavaScript proxy to the ActionScript mx.data.DataStore class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
...
function fdmsLibraryReady()
{
    productService = new DataService("Products");
    productService.setDataSource(new DataStore("Products"));
    products = new ArrayCollection();
    var token = productService.fill(products);
    token.addResponder(new AsyncResponder(productsResult, productsFault));
}
...
</script>
```

AsyncResponder

JavaScript proxy to the ActionScript AsyncResponder class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    productService = new DataService("Products");
    productService.setDataSource(new DataStore("Products"));
    products = new ArrayCollection();
    var token = productService.fill(products);
    token.addResponder(new AsyncResponder(productsResult, productsFault));
}
...
</script>
```

ArrayCollection

JavaScript proxy to the ActionScript ArrayCollection class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    productService = new DataService("Products");
    products = new ArrayCollection();
    var token = productService.fill(products);
    token.addResponder(new AsyncResponder(productsResult, productsFault));
}
...
</script>
```

Sort

JavaScript proxy to the ActionScript mx.collections.Sort class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    productService = new DataService("Products");
    products = new ArrayCollection();
    var token = productService.fill(products);
    token.addResponder(new AsyncResponder(productsResult, productsFault));
}

function productsResultHandler(event)
{
    var sort = new Sort();
    sort.setField([new SortField("sku", true), new SortField("color")]);
    products.setSort(sort);
    products.refresh();
}
...
</script>
```

SortField

JavaScript proxy to the ActionScript mx.collections.SortField class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    productService = new DataService("Products");
    products = new ArrayCollection();
    var token = productService.fill(products);
    token.addResponder(new AsyncResponder(productsResult, productsFault));
}

function productsResultHandler(event)
{
    var sort = new Sort();
    sort.setField([new SortField("sku", true), new SortField("color")]);
    products.setSort(sort);
    products.refresh();
}
...
</script>
```

Producer

JavaScript proxy to the ActionScript mx.messaging.Producer class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    producer = new Producer();
    producer.setDestination("dashboard_chat");
}

function sendChat(user, msg, color)
{
    var message = new AsyncMessage();
    var body = new Object();
    body.userId = user;
    body.msg = msg;
    body.color = color;
    message.setBody(body);
    producer.send(message);
}
...
</script>
```

Consumer

JavaScript proxy to the ActionScript mx.messaging.Consumer class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    consumer = new Consumer();
    consumer.setDestination("dashboard_chat");
    consumer.addEventListener("message", messageHandler);
    consumer.subscribe();
}

function messageHandler(event)
{
    body = event.getMessage().getBody();
    alert(body.userId + ":" + body.msg);
}
...
</script>
```

AsyncMessage

Description

JavaScript proxy to the ActionScript mx.messaging.AsyncMessage class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    producer = new Producer();
    producer.setDestination("dashboard_chat");
}

function sendChat(user, msg, color)
{
    var message = new AsyncMessage();
    var body = new Object();
    body.userId = user;
    body.msg = msg;
    body.color = color;
    message.setBody(body);
    producer.send(message);
}
...
</script>
```

ChannelSet

JavaScript proxy to the ActionScript mx.messaging.ChannelSet class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");

    var cs = new ChannelSet();
    cs.addChannel(new RTMPChannel("my-rtmp", "rtmp://localhost:2035"));
    cs.addChannel(new AMFChannel("my-polling-amf",
        "http://servername:8100/app/messagebroker/amfpolling"));
    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);
    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

RTMPChannel**Description**

JavaScript proxy to the ActionScript mx.messaging.channels.RTMPChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");
    var cs = new ChannelSet();
    cs.addChannel(new RTMPChannel("my-rtmp", "rtmp://servername:2035"));
    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);
    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

AMFChannel

JavaScript proxy to the ActionScript mx.messaging.channels.AMFChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");

    var cs = new ChannelSet();
    cs.addChannel(new AMFChannel("my-polling-
amf","http://servername:8100/app/messagebroker/amfpolling"));
    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);
    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

HTTPChannel

JavaScript proxy to the ActionScript mx.messaging.channels.HTTPChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");
    var cs = new ChannelSet();
    cs.addChannel(new HTTPChannel("my-
http","http://servername:8100/app/messagebroker/http"));
    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);
    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

SecureAMFChannel

JavaScript proxy to the ActionScript mx.messaging.channels.SecureAMFChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");
    var cs = new ChannelSet();
    cs.addChannel(new SecureAMFChannel("my-secure-
amf","https://servername:8100/app/messagebroker/secureamf"));

    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);

    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

SecureRTMPChannel

Description

JavaScript proxy to the ActionScript mx.messaging.channels.SecureRTMPChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");
    var cs = new ChannelSet();
    cs.addChannel(new SecureRTMPChannel("my-rtmps","rtmps://servername:443"));
    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);

    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

SecureHTTPChannel

JavaScript proxy to the ActionScript mx.messaging.channels.SecureHTTPChannel class. This class has the same API as the ActionScript 3.0 version.

Example

```
...
<script>
function fdmsLibraryReady()
{
    alert("Library Ready");
    var cs = new ChannelSet();
    cs.addChannel(new SecureHTTPChannel("my-secure-http",
    "https://servername:8100/app/messagebroker/securehttp"));
    cs.addChannel(new AMFChannel("my-polling-amf",
    "/app/messagebroker/amfpolling"));

    p = new Producer();
    p.setDestination("MyJMSTopic");
    p.setChannelSet(cs);

    c = new Consumer();
    c.setDestination("MyJMSTopic");
    c.addEventListener("message", messageHandler);
    c.setChannelSet(cs);
    c.subscribe();
}
...
</script>
```

FDMSLib JavaScript

The bridge provides a gateway between the JavaScript virtual machine and the ActionScript virtual machine for making calls to objects hosted by Flash Player. This bridge is designed specifically to handle various aspects of asynchronous calls that the Data Management Service and Messaging APIs require.

Example

```
/**
 *Once the FDMSBridge SWF file is loaded, this method is called.
 */
function initialize_FDMSBridge(typeData)
{
    for (var i=0; i<typeData.length; i++)
        FDMSBridge.addTypeInfo(typeData[i]);

    // setup ArrayCollection etc.
    ArrayCollection.prototype =
        FDMSBridge.getTypeFromName("mx.collections::ArrayCollection");
    DataService.prototype = FDMSBridge.getTypeFromName("mx.data::DataService");
    ...
    // now callback the page specific code indicating that initialization is
    // complete
    FDMSBridge.initialized();
}
```

Extending applications with factories

Adobe LiveCycle Data Services provides a factory mechanism that lets you plug in your own component creation and maintenance system to allow it to integrate with systems like EJB and Spring, which store components in their own namespace. You provide a class that implements the `flex.messaging.FlexFactory` interface. This class is used to create a `FactoryInstance` that corresponds to a component configured for a specific destination.

The factory mechanism

Remoting Service destinations and Data Management Service destinations both use Java classes that you write to integrate with Flex clients. By default, LiveCycle Data Services creates these instances. If they are application-scoped components, they are stored in the `ServletContext` attribute space using the destination's name as the attribute name. If you use session-scoped components, the components are stored in the `FlexSession`, also using the destination name as the attribute. If you specify an `attribute-id` element in the destination, you can control which attribute the component is stored in; this lets more than one destination share the same instance.

The following examples shows a destination definition that contains an `attribute-id` element:

```
<destination id="WeatherService">
  <properties>
    <source>weather.WeatherService</source>
    <scope>application</scope>
    <attribute-id>MyWeatherService</attribute-id>
  </properties>
</destination>
```

In this example, LiveCycle Data Services creates an instance of the class `weather.WeatherService` and stores it in the `ServletContext` object's set of attributes with the name `MyWeatherService`. If you define a different destination with the same `attribute-id` value and the same Java class, LiveCycle Data Services uses the same component instance.

LiveCycle Data Services provides a factory mechanism that lets you plug in your own component creation and maintenance system to LiveCycle Data Services so it integrates with systems like EJB and Spring, which store components in their own namespace. You provide a class that implements the `flex.messaging.FlexFactory` interface. You use this class to create a `FactoryInstance` that corresponds to a component configured for a specific destination. Then the component uses the `FactoryInstance` to look up the specific component to use for a given request. The `FlexFactory` implementation can access configuration attributes from a LiveCycle Data Services configuration file and also can access `FlexSession` and `ServletContext` objects. For more information, see the documentation for the `FlexFactory` class in the public LiveCycle Data Services Javadoc documentation.

After you implement a `FlexFactory` class, you can configure it to be available to destinations by placing a `factory` element in the `factories` element in the `services-config.xml` file, as the following example shows. A single `FlexFactory` instance is created for each LiveCycle Data Services web application. This instance can have its own configuration properties, although in this example, there are no required properties to configure the factory.

```
<factories>
  <factory id="spring" class="flex.samples.factories.SpringFactory"/>
</factories>
```

LiveCycle Data Services creates one `FlexFactory` instance for each `factory` element that you specify. LiveCycle Data Services uses this one global `FlexFactory` implementation for each destination that specifies that factory by its ID; the ID is identified by using a `factory` element in the `properties` section of the destination definition. For example, your `remoting-config.xml` file could have a destination similar to the following one:

```
<destination id="WeatherService">
  <properties>
    <factory>spring</factory>
    <source>weatherBean</source>
  </properties>
</destination>
```

When the `factory` element is encountered at start up, LiveCycle Data Services calls the `FlexFactory.createFactoryInstance()` method. That method gets the `source` value and any other attributes it expects in the configuration. Any attributes that you access from the `ConfigMap` parameter are marked as expected and do not cause a configuration error, so you can extend the default LiveCycle Data Services configuration in this manner. When LiveCycle Data Services requires an instance of the component for this destination, it calls the `FactoryInstance.lookup()` method to retrieve the individual instance for the destination.

Optionally, factory instances can take additional attributes. There are two places you can do this. When you define the factory initially, you can provide extra attributes as part of the factory definition. When you define an instance of that factory, you can also add your own attributes to the destination definition to be used in creating the factory instance.

The boldface text in the following example shows an attribute added to the factory definition:

```
<factories>
  <factory id="myFactoryId" class="myPackage.MyFlexFactory">
    <properties>
      <myfactoryattributename>
        myfactoryattributevalue
      </myfactoryattributename>
    </properties>
  </factory>
</factories>
```

You could use this type of configuration when you are integrating with the Spring Framework Java application framework to provide the Spring factory with a default path for initializing the Spring context that you use to look up all components for that factory. In the class that implements `FlexFactory`, you would include a call to retrieve the values of the `myfactoryattributename` from the `configMap` parameter to the `initialize()` method in the `FlexFactory` interface, as the following example shows:

```
public void initialize(String id, ConfigMap configMap) {
  System.out.println("***** MyFactory initialized with: " +
    configMap.getPropertyAsString("myfactoryattributename", "not set"));
}
```

The `initialize()` method in the previous example retrieves a string value where the first parameter is the name of the attribute, and the second parameter is the default value to use if that value is not set. For more information about the various calls to retrieve properties in the config map, see the documentation for the `flex.messaging.config.ConfigMap` class in the public LiveCycle Data Services Javadoc documentation. Each factory instance can add configuration attributes that are used when that factory instance is defined, as the following example shows:

```
<destination id="myDestination">
  <properties>
    <source>mypackage.MyRemoteClass</source>
    <factory>myFactoryId</factory>
    <myfactoryinstanceattribute>
      myfoobar2value
    </myfactoryinstanceattribute>
  </properties>
</destination>
```

In the `createFactoryInstance()` method as part of the FlexFactory implementation, you access the attribute for that instance of the factory, as the following example shows:

```
public FactoryInstance createFactoryInstance(String id, ConfigMap properties) {  
    System.out.println("**** MyFactoryInstance instance initialized with  
myfactoryinstanceattribute=" +  
    properties.getPropertyAsString("myfactoryinstanceattribute", "not set"));  
...  
}
```

The following example shows the source code of a Spring factory class:

```
package flex.samples.factories;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.web.context.support.WebApplicationContextUtils;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.NoSuchBeanDefinitionException;  
  
import flex.messaging.FactoryInstance;  
import flex.messaging.FlexFactory;  
import flex.messaging.config.ConfigMap;  
import flex.messaging.services.ServiceException;  
  
/**  
 * The FactoryFactory interface is implemented by factory components that provide  
 * instances to the data services messaging framework. To configure data services  
 * to use this factory, add the following lines to your services-config.xml  
 * file (located in the WEB-INF/flex directory of your web application).  
 *  
<pre>  
 *   <factories>  
 *     <factory id="spring" class="flex.samples.factories.SpringFactory" />  
 *   </factories>  
</pre>  
 *  
 * You also must configure the web application to use spring and must copy the spring.jar  
 * file into your WEB-INF/lib directory. To configure your app server to use Spring,  
 * you add the following lines to your WEB-INF/web.xml file:  
*<pre>  
*   <context-param>  
*     <param-name>contextConfigLocation</param-name>  
*     <param-value>/WEB-INF/applicationContext.xml</param-value>  
*   </context-param>  
*  
*   <listener>  
*     <listener-class>  
*       org.springframework.web.context.ContextLoaderListener</listener-class>  
*     </listener>  
*</pre>  
* Then you put your Spring bean configuration in WEB-INF/applicationContext.xml (as per the  
* line above). For example:  
*<pre>  
*   <?xml version="1.0" encoding="UTF-8"?>  
*   <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
*   "http://www.springframework.org/dtd/spring-beans.dtd">  
*
```

```
*  <beans>
*    <bean name="weatherBean" class="dev.weather.WeatherService" singleton="true"/>
*  </beans>
*  </pre>
* Now you are ready to define a Remoting Service destination that maps to this existing service.
* To do this you'd add this to your WEB-INF/flex/remoting-config.xml:
*
*<pre>
*  <destination id="WeatherService">
*    <properties>
*      <factory>spring</factory>
*      <source>weatherBean</source>
*    </properties>
*  </destination>
*</pre>
*/
public class SpringFactory implements FlexFactory
{
    private static final String SOURCE = "source";

    /**
     * This method can be used to initialize the factory itself.
     * It is called with configuration
     * parameters from the factory tag which defines the id of the factory.
     */
    public void initialize(String id, ConfigMap configMap) {}

    /**
     * This method is called when we initialize the definition of an instance
     * which will be looked up by this factory. It should validate that
     * the properties supplied are valid to define an instance.
     * Any valid properties used for this configuration must be accessed to
     * avoid warnings about unused configuration elements. If your factory
     * is only used for application scoped components, this method can simply
     * return a factory instance which delegates the creation of the component
     * to the FactoryInstance's lookup method.
     */
    public FactoryInstance createFactoryInstance(String id, ConfigMap properties)
    {
        SpringFactoryInstance instance = new SpringFactoryInstance(this, id, properties);
        instance.setSource(properties.getPropertyAsString(SOURCE, instance.getId()));
        return instance;
    } // end method createFactoryInstance()

    /**
     * Returns the instance specified by the source
     * and properties arguments. For the factory, this may mean
     * constructing a new instance, optionally registering it in some other
     * name space such as the session or JNDI, and then returning it
     * or it may mean creating a new instance and returning it.
     * This method is called for each request to operate on the
     * given item by the system so it should be relatively efficient.
     * <p>
     * If your factory does not support the scope property, it
     * report an error if scope is supplied in the properties
     * for this instance.
     * </p>

```

```
/*
public Object lookup(FactoryInstance inst)
{
    SpringFactoryInstance factoryInstance = (SpringFactoryInstance) inst;
    return factoryInstance.lookup();
}

static class SpringFactoryInstance extends FactoryInstance
{
    SpringFactoryInstance(SpringFactory factory, String id, ConfigMap properties)
    {
        super(factory, id, properties);
    }

    public String toString()
    {
        return "SpringFactory instance for id=" + getId() + " source=" + getSource() +
               " scope=" + getScope();
    }

    public Object lookup()
    {
        ApplicationContext appContext =
WebApplicationContextUtils.getWebApplicationContext(flex.messaging.FlexContext.getServletConfig()
.getServletContext());
        String beanName = getSource();

        try
        {
            return appContext.getBean(beanName);
        }
        catch (NoSuchBeanDefinitionException nexc)
        {
            ServiceException e = new ServiceException();
            String msg = "Spring service named '" + beanName + "' does not exist.";
            e.setMessage(msg);
            e.setRootCause(nexc);
            e.setDetails(msg);
        }
    }
}
```

```
        e.setCode("Server.Processing");
        throw e;
    }
    catch (BeansException bexc)
    {
        ServiceException e = new ServiceException();
        String msg = "Unable to create Spring service named '" + beanName + "' ";
        e.setMessage(msg);
        e.setRootCause(bexc);
        e.setDetails(msg);
        e.setCode("Server.Processing");
        throw e;
    }
}
}
```