Jersey 2.0 User Guide

Jersey 2.0 User Guide

Table of Contents

List of Tables

List of Examples

1. Preface

This is user guide for Jersey 2.0. We are trying to keep it up to date as we add new features. When reading the user guide, please consult also our Jersey API documentation [http://jersey.java.net/nonav/apidocs/2.0/org/glassfish/jersey/index.html] as an additional source of information about Jersey features and API.

If you would like to contribute to the guide or have questions on things not covered in our docs, please contact us atusers@jersey.java.net [mailto:users@jersey.java.net]. Similarly, in case you spot any errors in the Jersey documentation, please report them by filing a new issue in our Jersey JIRA Issue Tracker [http://java.net/jira/browse/JERSEY] under `docs` component. Please make sure to specify the version of the Jersey User Guide where the error has been spotted by selecting the proper value for the `Affected Version` field.

Text formatting conventions

First mention of any Jersey and JAX-RS API component in a section links to the API documentation of the referenced component. Any sub-sequent mentions of the component in the same chapter are rendered using a `monospace` font.

Emphasised font is used to a call attention to a newly introduce concept, when it first occurs in the text.

In some of the code listings, certain lines are too long to be displayed on one line for the available page width. In such case, the lines that exceed the available page width are broken up into multiple lines using a `'\'` at the end of each line to indicate that a break has been introduced to fit the line in the page. For example:

```
This is an overly long line that \
might not fit the available page \
width and had to be broken into \
multiple lines.

This line fits the page width.
```

Should read as:

```
This is an overly long line that might not fit the available page width and had to

This line fits the page width.
```

Chapter 1. Getting Started

This chapter provides a quick introduction on how to get started building RESTful services using Jersey. The example described here uses the lightweight Grizzly HTTP server. At the end of this chapter you will see how to implement equivalent functionality as a JavaEE web application you can deploy on any servlet container supporting Servlet 2.5 and higher.

1.1. Creating a New Project from Maven Archetype

If you want to depend on Jersey snapshot versions the following repository needs to be added to the pom:

```
<repository>
                <id>snapshot-repository.java.net</id>
                <name>Java.net Snapshot Repository for Maven</name>
                <url>https://maven.java.net/content/repositories/snapshots/</url>
                <layout>default</layout>
                </repository>
```

Now, to create a new Jersey project, based on Grizzly 2 container, from a maven archetype, execute the following in the directory where the new project should reside:

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2 -Darchetyp
```

Feel free to adjust the group id, package name and artifact id of your new project in the line above, or you can change it after it gets generated by updating the project pom.xml

1.2. Exploring the Newly Created Project

TODO

1.3. Customizing the JAX-RS Resource

TODO: instructions on how to make simple edits to the newly created resource

1.4. Running the Project

TODO: instructions on how to run the project

1.5. Creating a JavaEE Web Application

TODO

1.6. Exploring Other Jersey Examples

Jersey codebase contains a number of useful samples on how to use various JAX-RS and Jersey features. Please refer to the [TODO: Examples] section of this guide for more information on those.

Chapter 2. Jersey Modules and Dependencies

Jersey is built, assembled and installed using Maven. Jersey is deployed to the Java.Net maven repository at the following location: http://maven.java.net/ [https://maven.java.net/index.html]. The Jersey modules can be browsed at the following location: https://maven.java.net/content/repositories/releases/org/glassfish/jersey. Jars, Jar sources, Jar JavaDoc and samples are all available on the java.net maven repository.

An application depending on Jersey requires that it in turn includes the set of jars that Jersey depends on. Jersey has a pluggable component architecture so the set of jars required to be include in the class path can be different for each application.

All Jersey components are built using Java SE 6 compiler. It means, you will also need at least Java SE 6 to be able to compile and run your application.

Developers using maven are likely to find it easier to include and manage dependencies of their applications than developers using ant or other build technologies. This document will explain to both maven and non-maven developers how to depend on Jersey for their application. Ant developers are likely to find the Ant Tasks for Maven [http://maven.apache.org/ant-tasks/index.html] very useful.

The following table provides an overview of all Jersey modules and their dependencies with links to the respective binaries.

Table 2.1. Jersey modules and dependencies

Module	Dependencies	Description
	Core	
jersey-server [http://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=org.glassfish.jersey&a=jersey-server&v=2.0&e=pom]	jersey-commons [http://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=org.glassfish.jersey&a=jersey-commons&v=2.0&e=pom]	Base server functionality.
jersey-client	jersey-commons	Basic client functionality.
jersey-commons		Common functionality shared by client and server.
	Containers	
...	...	...

2.1. Sample Scenarios

2.1.1. Basic Server-Side Application

For a server side Jersey application you typically need to depend on jersey-server module to provide the basic functionality, then you may want to support JSON mapping and a standard JavaEE servlet container you would deploy your application to. So this would be the common set of dependencies for your project for this kind of scenario:

- jersey-server
- jersey-commons
- ...

Chapter 3. JAX-RS Application, Resources and Sub-Resources

This chapter presents an overview of the core JAX-RS concepts - resources and sub-resources.

The JAX-RS 2.0 JavaDoc can be found online here [http://jax-rs-spec.java.net/nonav/2.0/apidocs/index.html].

The JAX-RS 2.0 specification draft can be found online here [http://jcp.org/en/jsr/summary?id=339].

3.1. Root Resource Classes

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] have at least one method annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] or a resource method designator annotation such as @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html], @PUT [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PUT.html], @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DELETE.html]. Resource methods are methods of a resource class annotated with a resource method designator. This section shows how to use Jersey to annotate Java objects to create RESTful web services.

The following code example is a very simple example of a root resource class using JAX-RS annotations. The example code shown here is from one of the samples that ships with Jersey, the zip file of which can be found in the maven repository here [https://maven.java.net/content/repositories/releases/org/glassfish/jersey/examples/helloworld/2.0/].

Example 3.1. Simple hello world root resource class

```

package org.glassfish.jersey.examples.helloworld;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("helloworld")
public class HelloWorldResource {
    public static final String CLICHED_MESSAGE = "Hello Wo

@GET
@Produces("text/plain")
    public String getHello() {
        return CLICHED_MESSAGE;
    }
}

```

Let's look at some of the JAX-RS annotations used in this example.

3.1.1. @Path

The @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation. What makes JAX-RS so useful is that you can embed variables in the URIs.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation:

```
@Path("/users/{username}")
```

In this type of example, a user will be prompted to enter their name, and then a Jersey web service configured to respond to requests to this URI path template will respond. For example, if the user entered their username as "Galileo", the web service will respond to the following URL: `http://example.com/users/Galileo`

To obtain the value of the username variable the @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html] may be used on method parameter of a request method, for example:

Example 3.2. Specifying URI path parameter

```

@Path("/users/{username}")
public class UserResource {

@GET
@Produces("text/xml")
public String getUser(@PathParam("username") String us
...
}
}

```

If it is required that a user name must only consist of lower and upper case numeric characters then it is possible to declare a particular regular expression, which overrides the default regular expression, "[^/]+?", for example:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

In this type of example the username variable will only match user names that begin with one upper or lower case letter and zero or more alpha numeric characters and the underscore character. If a user name does not match that a 404 (Not Found) response will occur.

A @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] value may or may not begin with a '/', it makes no difference. Likewise, by default, a @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] value may or may not end in a '/', it makes no difference, and thus request URLs that end or do not end in a '/' will both be matched.

3.1.2. @GET, @PUT, @POST, @DELETE, ... (HTTP Methods)

@GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html], @PUT [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PUT.html], @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DELETE.html] and @HEAD [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HEAD.html] are *resource method designator* annotations defined by JAX-RS and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by which of the HTTP methods the resource is responding to.

The following example is an extract from the storage service sample that shows the use of the PUT method to create or update a storage container:

Example 3.3. PUT method

```

@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toStrin

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}

```

By default the JAX-RS runtime will automatically support the methods HEAD and OPTIONS, if not explicitly implemented. For HEAD the runtime will invoke the implemented GET method (if present) and ignore the response entity (if set). A response returned for the OPTIONS method depends on the requested media type defined in the 'Accept' header. The OPTIONS method can return a response with a set of supported resource methods in the 'Allow' header or return a WADL [http://wadl.java.net/] document. See wadl section for more information.

3.1.3. @Produces

The @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain". @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] can be applied at both the class and method levels. Here's an example:

Example 3.4. Specifying output MIME type

```

@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}

```

The `doGetAsPlainText` method defaults to the MIME type of the @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation at the class level. The `doGetAsHtml` method's @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] annotation overrides the class-level @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] setting, and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more that one MIME media type then the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically the Accept header of the HTTP request declares what is most acceptable. For example if the Accept header is "`Accept: text/plain`" then the `doGetAsPlainText` method will be invoked. Alternatively if the Accept header is " `Accept: text/plain;q=0.9, text/html`", which declares that the client can accept media types of "text/plain" and "text/html" but prefers the latter, then the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] declaration, for example:

Example 3.5. Using multiple output MIME types

```

@GET
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}

```

The `doGetAsXmlOrJson` method will get invoked if either of the media types "application/xml" and "application/json" are acceptable. If both are equally acceptable then the former will be chosen because it occurs first.

Optionally, server can also specify the quality factor for individual media types. These are considered if several are equally acceptable by the client. For example:

Example 3.6. Server-side content negotiation

```

            @GET
            @Produces({"application/xml; qs=0.9", "application/jso
            public String doGetAsXmlOrJson() {
                ...
            }

```

In the above sample, if client accepts both "application/xml" and "application/json" (equally), then a server always sends "application/json", since "application/xml" has a lower quality factor.

The examples above refers explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors, see the constant field values of MediaType [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/MediaType.html].

3.1.4. @Consumes

The @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] annotation is used to specify the MIME media types of representations that can be consumed by a resource. The above example can be modified to set the cliched message as follows:

Example 3.7. Specifying input MIME type

```
@POST
            @Consumes("text/plain")
            public void postClichedMessage(String message) {
                // Store the message
            }

```

In this example, the Java method will consume representations identified by the MIME media type "text/plain". Notice that the resource method returns void. This means no representation is returned and response with a status code of 204 (No Content) will be returned to the client.

@Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] can be applied at both the class and the method levels and more than one media type may be declared in the same @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] declaration.

3.2. Parameter Annotations (@*Param)

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. One of the previous examples presented the use of @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html] to extract a path parameter from the path component of the request URL that matched the path declared in @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html].

@QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html] is used to extract query parameters from the Query component of the request URL. The following example is an extract from the sparklines sample:

Example 3.8. Query parameters

```

                @Path("smooth")
                @GET
                public Response smooth(
                    @DefaultValue("2") @QueryParam("step") int step,
                    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
                    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
                    @DefaultValue("true") @QueryParam("last-m") boolean hasLas
                    @DefaultValue("blue") @QueryParam("min-color") ColorParam
                    @DefaultValue("green") @QueryParam("max-color") ColorParam
                    @DefaultValue("red") @QueryParam("last-color") ColorParam
                    ...
                }

```

If a query parameter "step" exists in the query component of the request URI then the "step" value will be will extracted and parsed as a 32 bit signed integer and assigned to the step method parameter. If "step" does not exist then a default value of 2, as declared in the @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] annotation, will be assigned to the step method parameter. If the "step" value cannot be parsed as a 32 bit signed integer then a HTTP 404 (Not Found) response is returned. User defined Java types such as `ColorParam` may be used, which as implemented as follows:

Example 3.9. Custom Java type for consuming request parameters

```

                public class ColorParam extends Color {

                    public ColorParam(String s) {
                        super(getRGB(s));
                    }

                    private static int getRGB(String s) {
                        if (s.charAt(0) == '#') {
                            try {
                                Color c = Color.decode("0x" + s.substring(1));
                                return c.getRGB();
                            } catch (NumberFormatException e) {
                                throw new WebApplicationException(400);
                            }
                        } else {
                            try {
                                Field f = Color.class.getField(s);
                                return ((Color)f.get(null)).getRGB();
                            } catch (Exception e) {
                                throw new WebApplicationException(400);
                            }
                        }
                    }
                }

```

In general the Java type of the method parameter may:

1. Be a primitive type;

2. Have a constructor that accepts a single `String` argument;

3. Have a static method named `valueOf` or `fromString` that accepts a single `String` argument (see, for example, `Integer.valueOf(String)` and `java.util.UUID.fromString(String)`);

4. Have a registered implementation of `javax.ws.rs.ext.ParamConverterProvider` JAX-RS extension SPI that returns a `javax.ws.rs.ext.ParamConverter` instance capable of a "from string" conversion for the type. or

5. Be `List<T>`, `Set<T>` or `SortedSet<T>`, where `T` satisfies 2 or 3 above. The resulting collection is read-only.

Sometimes parameters may contain more than one value for the same name. If this is the case then types in 5) may be used to obtain all values.

If the @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] is not used in conjunction with @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html] and the query parameter is not present in the request then value will be an empty collection for`List`, `Set` or `SortedSet`, `null` for other object types, and the Java-defined default for primitive types.

The @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html] and the other parameter-based annotations, @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html], @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html], @CookieParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/CookieParam.html], @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] obey the same rules as @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html]. @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html] extracts information from URL path segments. @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html] extracts information from the HTTP headers. @CookieParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/CookieParam.html] extracts information from the cookies declared in cookie related HTTP headers.

@FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] is slightly special because it extracts information from a request representation that is of the MIME media type `"application/x-www-form-urlencoded"` and conforms to the encoding specified by HTML forms, as described here. This parameter is very useful for extracting information that is POSTed by HTML forms, for example the following extracts the form parameter named "name" from the POSTed form data:

Example 3.10. Processing POSTed HTML form

```

@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}

```

If it is necessary to obtain a general map of parameter name to values then, for query and path parameters it is possible to do the following:

Example 3.11. Obtaining general map of URI path and/or query parameters

```

                        @GET
                        public String get(@Context UriInfo ui) {
                            MultivaluedMap<String, String> queryParams = ui.getQue
                            MultivaluedMap<String, String> pathParams = ui.getPath
                        }

```

For header and cookie parameters the following:

Example 3.12. Obtaining general map of header parameters

```

                        @GET
                        public String get(@Context HttpHeaders hh) {
                            MultivaluedMap<String, String> headerParams = hh.getRe
                            Map<String, Cookie> pathParams = hh.getCookies();
                        }

```

In general @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] can be used to obtain contextual Java types related to the request or response.

Because form parameters (unlike others) are part of the message entity, it is possible to do the following:

Example 3.13. Obtaining general map of form parameters

```

                        @POST
                        @Consumes("application/x-www-form-urlencoded")
                        public void post(MultivaluedMap<String, String> formParams
                            // Store the message
                        }

```

I.e. you don't need to use the @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation.

Another kind of injection is the @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] which allows to inject the parameters described above into a single bean. A bean annotated with @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] containing any fields and appropriate `*param` annotation(like @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html]) will be initialized with corresponding request values in expected way as if these fields were in the resource class. Then instead of injecting request values like path param into a constructor parameters or class fields the @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] can be used to inject such a bean into a resource or resource method. The @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] is used this way to aggregate more request parameters into a single bean.

Example 3.14. Example of the bean which will be used as @BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html]

```

                public class MyBeanParam {
                    @PathParam("p")
                    private String pathParam;

                    @MatrixParam("m")
                    @Encoded
                    @DefaultValue("default")
                    private String matrixParam;

                    @HeaderParam("header")
                    private String headerParam;

                    private String queryParam;

                    public MyBeanParam(@QueryParam("q") String queryParam) {
                        this.queryParam = queryParam;
                    }

                    public String getPathParam() {
                        return pathParam;
                    }
                    ...
                }

```

Example 3.15. Injection of MyBeanParam as a method parameter:

```

                @POST
                public void post(@BeanParam MyBeanParam beanParam, String enti
                    final String pathParam = beanParam.getPathParam(); // cont
                    ...
                }

```

The example shows aggregation of injections @PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html], @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/QueryParam.html] @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html] and @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html] into one single bean. The rules for injections inside the bean are the same as described above for these injections. The @DefaultValue [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DefaultValue.html] is used to define the default value for matrix parameter matrixParam. Also the @Encoded [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Encoded.html] annotation has the same behaviour as if it were used for injection in the resource method directly. Injecting the bean parameter into @Singleton resource class fields is not allowed (injections into method parameter must be used instead).

@BeanParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/BeanParam.html] can contain all parameters injections injections (@PathParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/PathParam.html], @QueryParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/

javax/ws/rs/QueryParam.html], @MatrixParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/MatrixParam.html], @HeaderParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/HeaderParam.html], @CookieParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/CookieParam.html], @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html]). More beans can be injected into one resource or method parameters even if they inject the same request values. For example the following is possible:

Example 3.16. Injection of more beans into one resource methods:

```

@POST
public void post(@BeanParam MyBeanParam beanParam, @BeanParam
String entity) {
    // beanParam.getPathParam() == pathParam
    ...
}

```

3.3. Sub-resources

@Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may be used on classes and such classes are referred to as root resource classes. @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may also be used on methods of root resource classes. This enables common functionality for a number of resources to be grouped together and potentially reused.

The first way @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may be used is on resource methods and such methods are referred to as *sub-resource methods*. The following example shows the method signatures for a root resource class from the jmaki-backend sample:

Example 3.17. Sub-resource methods

```

@Singleton
@Path("/printers")
public class PrintersResource {

    @GET
    @Produces({"application/json", "application/xml"})
    public WebResourceList getMyResources() { ... }

    @GET @Path("/list")
    @Produces({"application/json", "application/xml"})
    public WebResourceList getListOfPrinters() { ... }

    @GET @Path("/jMakiTable")
    @Produces("application/json")
    public PrinterTableModel getTable() { ... }

    @GET @Path("/jMakiTree")
    @Produces("application/json")
    public TreeModel getTree() { ... }

    @GET @Path("/ids/{printerid}")
    @Produces({"application/json", "application/xml"})
    public Printer getPrinter(@PathParam("printerid") Stri

    @PUT @Path("/ids/{printerid}")
    @Consumes({"application/json", "application/xml"})
    public void putPrinter(@PathParam("printerid") String

    @DELETE @Path("/ids/{printerid}")
    public void deletePrinter(@PathParam("printerid") Stri
}

```

If the path of the request URL is "printers" then the resource methods not annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] will be selected. If the request path of the request URL is "printers/list" then first the root resource class will be matched and then the sub-resource methods that match "list" will be selected, which in this case is the sub-resource method`getListOfPrinters`. So, in this example hierarchical matching on the path of the request URL is performed.

The second way @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] may be used is on methods **not** annotated with resource method designators such as @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] or @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html]. Such methods are referred to as *sub-resource locators*. The following example shows the method signatures for a root resource class and a resource class from the optimistic-concurrency sample:

Example 3.18. Sub-resource locators

```

@Path("/item")
public class ItemResource {
    @Context UriInfo uriInfo;

    @Path("content")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }

    @GET
    @Produces("application/xml")
        public Item get() { ... }
    }
}

public class ItemContentResource {

    @GET
    public Response get() { ... }

    @PUT
    @Path("{version}")
    public void put(@PathParam("version") int version,
                    @Context HttpHeaders headers,
                    byte[] in) {
        ...
    }
}

```

The root resource class `ItemResource` contains the sub-resource locator method `getItemContentResource` that returns a new resource class. If the path of the request URL is "item/content" then first of all the root resource will be matched, then the sub-resource locator will be matched and invoked, which returns an instance of the `ItemContentResource` resource class. Sub-resource locators enable reuse of resource classes. A method can be annotated with the @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] annotation with empty path (`@Path("/")` or `@Path("")`) which means that the sub resource locator is matched for the path of the enclosing resource (without sub-resource path).

Example 3.19. Sub-resource locators with empty path

```

@Path("/item")
public class ItemResource {

    @Path("/")
    public ItemContentResource getItemContentResource() {
        return new ItemContentResource();
    }
}

```

In the example above the sub-resource locator method `getItemContentResource` is matched for example for request path "/item/locator" or even for only "/item".

In addition the processing of resource classes returned by sub-resource locators is performed at runtime thus it is possible to support polymorphism. A sub-resource locator may return different sub-types depending on the request (for example a sub-resource locator could return different sub-types dependent on the role of the principle that is authenticated). So for example the following sub resource locator is valid:

Example 3.20. Sub-resource locators returning sub-type

```

@Path("/item")
public class ItemResource {

    @Path("/")
    public Object getItemContentResource() {
        return new AnyResource();
    }
}

```

Note that the runtime will not manage the life-cycle or perform any field injection onto instances returned from sub-resource locator methods. This is because the runtime does not know what the life-cycle of the instance is. If it is required that the runtime manages the sub-resources as standard resources the `Class` should be returned as shown in the following example:

Example 3.21. Sub-resource locators created from classes

```

import javax.inject.Singleton;

@Path("/item")
public class ItemResource {
    @Path("content")
    public Class<ItemContentSingletonResource> getItemCont
        return ItemContentSingletonResource.class;
    }
}

    @Singleton
    public class ItemContentSingletonResource {
        // this class is managed in the singleton life cyc
    }
}

```

JAX-RS resources are managed in per-request scope by default which means that new resource is created for each request. In this example the `javax.inject.Singleton` annotation says that the resource will be managed as singleton and not in request scope. The sub-resource locator method returns a class which means that the runtime will managed the resource instance and its life-cycle. If the method would return instance instead, the `Singleton` annotation would have no effect and the returned instance would be used.

The sub resource locator can also return a *programmatic resource model*. See resource builder section for information of how the programmatic resource model is constructed. The following example shows very simple resource returned from the sub-resource locator method.

Example 3.22. Sub-resource locators returning resource model

```

import org.glassfish.jersey.server.model.Resource;

@Path("/item")
public class ItemResource {

    @Path("content")
    public Resource getItemContentResource() {
        return Resource.from(ItemContentSingletonResource.
    }
}

```

The code above has exactly the same effect as previous example. `Resource` is a resource simple resource constructed from `ItemContentSingletonResource`. More complex programmatic resource can be returned as long they are valid resources.

3.4. Life-cycle of Root Resource Classes

By default the life-cycle of root resource classes is per-request which, namely that a new instance of a root resource class is created every time the request URI path matches the root resource. This makes for a very natural programming model where constructors and fields can be utilized (as in the previous section showing the constructor of the `SparklinesResource` class) without concern for multiple concurrent requests to the same resource.

In general this is unlikely to be a cause of performance issues. Class construction and garbage collection of JVMs has vastly improved over the years and many objects will be created and discarded to serve and process the HTTP request and return the HTTP response.

Instances of singleton root resource classes can be declared by an instance of Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html].

Jersey supports two further life-cycles using Jersey specific annotations.

Table 3.1.

Scope	**Annotation**	**Annotation full class name**	**Description**
Request scope	@RequestScoped (or none)	org.glassfish.jersey.process.internal.RequestScoped	*Default lifecycle* (applied when no annotation is present). In this scope the resource instance is created for each new request and used for processing of this request. If the resource is used more than one time in the request processing, always the same instance will be used. This can

Scope	Annotation	Annotation full class name	Description
			happen when a resource is a sub resource is returned more times during the matching. In this situation only on instance will server the requests.
Per-lookup scope	@PerLookup	org.glassfish.jersey.process.internal.RequestScoped	In this scope the resource instance is created every time it is needed for the processing even it handles the same request.
Singleton	@Singleton	javax.inject.Singleton	In this scope there is only one instance per jax-rs application. Singleton resource can be either annotated with @Singleton and its class can be registered using the instance of Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html]. You can also create singletons by registering singleton instances into Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html].

3.5. Rules of Injection

Previous sections have presented examples of annotated types, mostly annotated method parameters but also annotated fields of a class, for the injection of values onto those types.

This section presents the rules of injection of values on annotated types. Injection can be performed on fields, constructor parameters, resource/sub-resource/sub-resource locator method parameters and bean setter methods. The following presents an example of all such injection cases:

Example 3.23. Injection

```

                    @Path("id: \d+")
                    public class InjectedResource {
                        // Injection onto field
                        @DefaultValue("q") @QueryParam("p")
                        private String p;

                        // Injection onto constructor parameter
                        public InjectedResource(@PathParam("id") int id) { ...

                        // Injection onto resource method parameter
                        @GET
                        public String get(@Context UriInfo ui) { ... }

                        // Injection onto sub-resource resource method paramet
                        @Path("sub-id")
                        @GET
                        public String get(@PathParam("sub-id") String id) { ..

                        // Injection onto sub-resource locator method paramete
                        @Path("sub-id")
                        public SubResource getSubResource(@PathParam("sub-id")

                        // Injection using bean setter method
                        @HeaderParam("X-header")
                        public void setHeader(String header) { ... }
                    }

```

There are some restrictions when injecting on to resource classes with a life-cycle of singleton scope. In such cases the class fields or constructor parameters cannot be injected with request specific parameters. So, for example the following is not allowed.

Example 3.24. Wrong injection into a singleton scope

```

                    @Path("resource")
                    @Singleton
                    public static class MySingletonResource {

                        @QueryParam("query")
                        String param; // WRONG: initialization of application
                                      // inject request specific parameters in

                        @GET
                        public String get() {
                            return "query param: " + param;
                        }
                    }

```

The example above will cause validation failure during application initialization as singleton resources cannot inject request specific parameters. The same example would fail if the query parameter would be injected into constructor parameter of such a singleton. In other words, if you wish one resource instance to server more requests (in the same time) it cannot be bound to a specific request parameter.

The exception exists for specific request objects which can injected even into constructor or class fields. For these objects the runtime will inject proxies which are able to simultaneously server more request. These request objects are `HttpHeaders`, `Request`, `UriInfo`, `SecurityContext`. These proxies can be injected using the @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation. The following example shows injection of proxies into the singleton resource class.

Example 3.25. Injection of proxies into singleton

```

@Path("resource")
@Singleton
public static class MySingletonResource {
    @Context
    Request request; // this is ok: the proxy of Request w

    public MySingletonResource(@Context SecurityContext se
        // this is ok too: the proxy of SecurityContext wi
    }

    @GET
    public String get() {
        return "query param: " + param;
    }
}

```

To summarize the injection can be done into the following constructs:

Table 3.2.

Java construct	Description
Class fields	Inject value directly into the field of the class. The field can be private and must not be final. Cannot be used in Singleton scope except proxiable types mentioned above.
Constructor parameters	The constructor will be invoked with injected values. If more constructors exists the one with the most injectable parameters will be invoked. Cannot be used in Singleton scope except proxiable types mentioned above.
Resource methods	The resource methods (these annotated with @GET, @POST, ...) can contain parameters that can be injected when the resource method is executed. Can be used in any scope.
Sub resource locators	The sub resource locators (methods annotated with @Path but not @GET, @POST, ...) can contain parameters that can be injected when the resource method is executed. Can be used in any scope.

Setter methods	Instead of injecting values directly into field the value can be injected into the setter method which will initialize the field. This injection can be used only with @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html] annotation. This means it cannot be used for example for injecting of query params but it can be used for injections of request. The setters will be called after the object creation and only once. The name of the method does not necessary have a setter pattern. Cannot be used in Singleton scope except proxiable types mentioned above.

The following example shows all possible java constructs into which the values can be injected.

Example 3.26. Example of possible injections

```

                    @Path("resource")
                    public static class SummaryOfInjectionsResource {
                        @QueryParam("query")
                        String param; // injection into a class field


                        @GET
                        public String get(@QueryParam("query") String methodQu
                            // injection into a resource method parameter
                            return "query param: " + param;
                        }

                        @Path("sub-resource-locator")
                        public Class<SubResource> subResourceLocator(@QueryPar
                            // injection into a sub resource locator parameter
                            return SubResource.class;
                        }

                        public SummaryOfInjectionsResource(@QueryParam("query"
                            // injection into a constructor parameter
                        }


                        @Context
                        public void setRequest(Request request) {
                            // injection into a setter method
                            System.out.println(request != null);
                        }
                    }

                    public static class SubResource {
                        @GET
                        public String get() {
                            return "sub resource";
                        }
                    }

```

The @FormParam [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/FormParam.html] annotation is special and may only be utilized on resource and sub-resource methods. This is because it extracts information from a request entity.

3.6. Use of @Context

Previous sections have introduced the use of @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html]. Chapter 5 of the JAX-RS specification presents all the standard JAX-RS Java types that may be used with @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html].

When deploying a JAX-RS application using servlet then ServletConfig [http://docs.oracle.com/javaee/5/api/javax/servlet/ServletConfig.html], ServletContext [http://docs.oracle.com/

javaee/5/api/javax/servlet/ServletContext.html], HttpServletRequest [http://docs.oracle.com/javaee/5/api/javax/servlet/http/HttpServletRequest.html] and HttpServletResponse [http://docs.oracle.com/javaee/5/api/javax/servlet/http/HttpServletResponse.html] are available using @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html].

3.7. Programmatic resource model

Resources can be constructed from classes or instances but also can be constructed from a programmatic resource model. Every resource created from from resource classes can also be constructed using the programmatic resource builder api. See resource builder section for more information.

Chapter 4. Representations and Responses

4.1. Representations and Java Types

Previous sections on @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] and @Consumes [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Consumes.html] referred to MIME media types of representations and showed resource methods that consume and produce the Java type String for a number of different media types. However, `String` is just one of many Java types that are required to be supported by JAX-RS implementations.

Java types such as`byte[]`, `java.io.InputStream`, `java.io.Reader` and `java.io.File` are supported. In addition JAXB beans are supported. Such beans are `JAXBElement` or classes annotated with @XmlRootElement [http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/XmlRootElement.html] or@XmlType [http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/XmlType.html]. The samples jaxb and json-from-jaxb show the use of JAXB beans.

Unlike method parameters that are associated with the extraction of request parameters, the method parameter associated with the representation being consumed does not require annotating. A maximum of one such unannotated method parameter may exist since there may only be a maximum of one such representation sent in a request.

The representation being produced corresponds to what is returned by the resource method. For example JAX-RS makes it simple to produce images that are instance of `File` as follows:

Example 4.1. Using `File` with a specific MIME type to produce a response

```
@GET
                    @Path("/images/{image}")
                    @Produces("image/*")
                    public Response getImage(@PathParam("image") String image)
                    File f = new File(image);

                    if (!f.exists()) {
                    throw new WebApplicationException(404);
                    }

                    String mt = new MimetypesFileTypeMap().getContentType(f);
                    return Response.ok(f, mt).build();
                    }

```

A `File` type can also be used when consuming, a temporary file will be created where the request entity is stored.

The `Content-Type` (if not set, see next section) can be automatically set from the MIME media types declared by @Produces [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Produces.html] if the most acceptable media type is not a wild card (one that contains a *, for example "application/" or "/*"). Given the following method, the most acceptable MIME type is used when multiple output MIME types allowed:

```
@GET
```

```
                @Produces({"application/xml", "application/json"})
                public String doGetAsXmlOrJson() {
                ...
                }

```

If "application/xml" is the most acceptable then the `Content-Type` of the response will be set to "application/xml".

4.2. Building Responses

Sometimes it is necessary to return additional information in response to a HTTP request. Such information may be built and returned using Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html] and Response.ResponseBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.ResponseBuilder.html]. For example, a common RESTful pattern for the creation of a new resource is to support a POST request that returns a 201 (Created) status code and a `Location` header whose value is the URI to the newly created resource. This may be achieved as follows:

Example 4.2. Returning 201 status code and adding `Location` header in response to POST request

```
@POST
                    @Consumes("application/xml")
                    public Response post(String content) {
                    URI createdUri = ...
                    create(content);
                    return Response.created(createdUri).build();
                    }

```

In the above no representation produced is returned, this can be achieved by building an entity as part of the response as follows:

Example 4.3. Adding an entity body to a custom response

```
@POST
                    @Consumes("application/xml")
                    public Response post(String content) {
                    URI createdUri = ...
                    String createdContent = create(content);
                    return Response.created(createdUri).entity(createdContent)
                    }

```

Response building provides other functionality such as setting the entity tag and last modified date of the representation.

4.3. WebApplicationException and Mapping Exceptions to Responses

Previous sections have shown how to return HTTP responses and it is possible to return HTTP errors using the same mechanism. However, sometimes when programming in Java it is more natural to use exceptions for HTTP errors.

The following example shows the throwing of a `NotFoundException` from the bookmark sample:

Example 4.4. Throwing Jersey specific exceptions to control response

```
@Path("items/{itemid}/")
                    public Item getItem(@PathParam("itemid") String itemid) {
                    Item i = getItems().get(itemid);
                    if (i == null)
                    throw new NotFoundException("Item, " + itemid + ", is not

                    return i;
                    }

```

This exception is a Jersey specific exception that extends WebApplicationException [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html] and builds a HTTP response with the 404 status code and an optional message as the body of the response:

Example 4.5. Jersey specific exception implementation

```
public class NotFoundException extends WebApplicationException {

                    /**
                    * Create a HTTP 404 (Not Found) exception.
                    */
                    public NotFoundException() {
                    super(Responses.notFound().build());
                    }

                    /**
                    * Create a HTTP 404 (Not Found) exception.
                    * @param message the String that is the entity of the 404
                    */
                    public NotFoundException(String message) {
                    super(Response.status(Responses.NOT_FOUND).
                    entity(message).type("text/plain").build());
                    }

                    }

```

In other cases it may not be appropriate to throw instances of WebApplicationException [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html], or classes that extend WebApplicationException [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/WebApplicationException.html], and instead it may be preferable to map an existing exception to a response. For such cases it is possible to use the ExceptionMapper<E extends Throwable> [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ExceptionMapper.html] interface. For example, the following maps the EntityNotFoundException [http://docs.oracle.com/javaee/5/api/javax/persistence/EntityNotFoundException.html] to a HTTP 404 (Not Found) response:

Example 4.6. Mapping generic exceptions to responses

```
@Provider
                    public class EntityNotFoundMapper implements
                    ExceptionMapper<javax.persistence.EntityNotFoundException>
                    public Response toResponse(javax.persistence.EntityNotFoun
                    return Response.status(404).
                    entity(ex.getMessage()).
                    type("text/plain").
                    build();
                    }
                    }

```

The above class is annotated with @Provider [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/Provider.html], this declares that the class is of interest to the JAX-RS runtime. Such a class may be added to the set of classes of the Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] instance that is configured. When an application throws an EntityNotFoundException [http://docs.oracle.com/javaee/5/api/javax/persistence/EntityNotFoundException.html] the `toResponse` method of the `EntityNotFoundMapper` instance will be invoked.

4.4. Conditional GETs and Returning 304 (Not Modified) Responses

Conditional GETs are a great way to reduce bandwidth, and potentially server-side performance, depending on how the information used to determine conditions is calculated. A well-designed web site may return 304 (Not Modified) responses for the many of the static images it serves.

JAX-RS provides support for conditional GETs using the contextual interface Request [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html].

The following example shows conditional GET support from the sparklines sample:

Example 4.7. Conditional GET support

```
public SparklinesResource(
                    @QueryParam("d") IntegerList data,
                    @DefaultValue("0,100") @QueryParam("limits") Interval limi
                    @Context Request request,
                    @Context UriInfo ui) {
                    if (data == null)
                    throw new WebApplicationException(400);

                    this.data = data;

                    this.limits = limits;

                    if (!limits.contains(data))
                    throw new WebApplicationException(400);

                    this.tag = computeEntityTag(ui.getRequestUri());
                    if (request.getMethod().equals("GET")) {
                    Response.ResponseBuilder rb = request.evaluatePrecondition
                    if (rb != null)
                    throw new WebApplicationException(rb.build());
                    }
                    }

```

The constructor of the `SparklinesResouce` root resource class computes an entity tag from the request URI and then calls the request.evaluatePreconditions [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] with that entity tag. If a client request contains an `If-None-Match` header with a value that contains the same entity tag that was calculated then the evaluatePreconditions [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] returns a pre-filled out response, with the 304 status code and entity tag set, that may be built and returned. Otherwise, evaluatePreconditions [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Request.html#evaluatePreconditions(javax.ws.rs.core.EntityTag)] returns `null` and the normal response can be returned.

Notice that in this example the constructor of a resource class can be used perform actions that may otherwise have to be duplicated to invoked for each resource method.

Chapter 5. URIs and Links

5.1. Building URIs

A very important aspects of REST is hyperlinks, URIs, in representations that clients can use to transition the Web service to new application states (this is otherwise known as "hypermedia as the engine of application state"). HTML forms present a good example of this in practice.

Building URIs and building them safely is not easy with java.net.URI [http://docs.oracle.com/javase/1.5.0/docs/api/java/net/URI.html], which is why JAX-RS has the UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] class that makes it simple and easy to build URIs safely.

UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] can be used to build new URIs or build from existing URIs. For resource classes it is more than likely that URIs will be built from the base URI the web service is deployed at or from the request URI. The class UriInfo [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html] provides such information (in addition to further information, see next section).

The following example shows URI building with UriInfo [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html] and UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] from the bookmark sample:

Example 5.1. URI building

```
@Path("/users/")
                    public class UsersResource {

                    @Context UriInfo uriInfo;

                    ...

                    @GET
                    @Produces("application/json")
                    public JSONArray getUsersAsJsonArray() {
                    JSONArray uriArray = new JSONArray();
                    for (UserEntity userEntity : getUsers()) {
                    UriBuilder ub = uriInfo.getAbsolutePathBuilder();
                    URI userUri = ub.
                    path(userEntity.getUserid()).
                    build();
                    uriArray.put(userUri.toASCIIString());
                    }
                    return uriArray;
                    }
                    }

```

UriInfo [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html] is obtained using the @Context annotation, and in this particular example injection onto the field of the root resource class is performed, previous examples showed the use of @Context on resource method parameters.

UriInfo [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html] can be used to obtain URIs and associated UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/

UriBuilder.html] instances for the following URIs: the base URI the application is deployed at; the request URI; and the absolute path URI, which is the request URI minus any query components.

The `getUsersAsJsonArray` method constructs a JSONArrray where each element is a URI identifying a specific user resource. The URI is built from the absolute path of the request URI by calling UriInfo.getAbsolutePathBuilder() [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriInfo.html#getAbsolutePathBuilder()]. A new path segment is added, which is the user ID, and then the URI is built. Notice that it is not necessary to worry about the inclusion of '/' characters or that the user ID may contain characters that need to be percent encoded. UriBuilder takes care of such details.

UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] can be used to build/replace query or matrix parameters. URI templates can also be declared, for example the following will build the URI "http://localhost/segment?name=value":

Example 5.2. Building URIs using query parameters

```
UriBuilder.fromUri("http://localhost/").
                    path("{a}").
                    queryParam("name", "{value}").
                    build("segment", "value");

```

Chapter 6. Deploying a RESTful Web Service

JAX-RS provides a deployment agnostic abstract class Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] for declaring root resource and provider classes, and root resource and provider singleton instances. A Web service may extend this class to declare root resource and provider classes. For example,

Example 6.1. Deployment agnostic application model

```
public class MyApplication extends Application {
                public Set<Class<?>> getClasses() {
                Set<Class<?>> s = new HashSet<Class<?>>();
                s.add(HelloWorldResource.class);
                return s;
                }
                }

```

Alternatively it is possible to reuse one of Jersey's implementations that scans for root resource and provider classes given a classpath or a set of package names. Such classes are automatically added to the set of classes that are returned by`getClasses`. For example, the following scans for root resource and provider classes in packages "org.foo.rest", "org.bar.rest" and in any sub-packages of those two:

Example 6.2. Reusing Jersey implementation in your custom application model

```
public class MyApplication extends PackagesResourceConfig {
                public MyApplication() {
                super("org.foo.rest;org.bar.rest");
                }
                }

```

There are multiple deployment options for the class that implements Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] interface in the Servlet 3.0 container. For simple deployments, no `web.xml` is needed at all. Instead, an @ApplicationPath [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ApplicationPath.html] annotation can be used to annotate the user defined application class and specify the the base resource URI of all application resources:

Example 6.3. Deployment of a JAX-RS application using `@ApplicationPath` with Servlet 3.0

```
@ApplicationPath("resources")
                public class MyApplication extends PackagesResourceConfig {
                public MyApplication() {
                super("org.foo.rest;org.bar.rest");
                }
                ...
                }

```

You also need to set maven-war-plugin attribute failOnMissingWebXml [http://maven.apache.org/plugins/maven-war-plugin/war-mojo.html#failOnMissingWebXml] to false in pom.xml when building .war without web.xml file using maven:

Example 6.4. Configuration of maven-war-plugin in `pom.xml` with Servlet 3.0

```
<plugins>
                ...
                <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.1.1</version>
                <configuration>
                <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
                </plugin>
                ...
                </plugins>
```

Another deployment option is to declare JAX-RS application details in the`web.xml`. This is usually suitable in case of more complex deployments, e.g. when security model needs to be properly defined or when additional initialization parameters have to be passed to Jersey runtime. JAX-RS 1.1 specifies that a fully qualified name of the class that implements Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] may be declared in the `<servlet-name>` element of the JAX-RS application's `web.xml`. This is supported in a Web container implementing Servlet 3.0 as follows:

Example 6.5. Deployment of a JAX-RS application using `web.xml` with Servlet 3.0

```
<web-app>
                <servlet>
                <servlet-name>org.foo.rest.MyApplication</servlet-name>
                </servlet>
                ...
                <servlet-mapping>
                <servlet-name>org.foo.rest.MyApplication</servlet-name>
                <url-pattern>/resources</url-pattern>
                </servlet-mapping>
                ...
                </web-app>
```

Note that the `<servlet-class>` element is omitted from the servlet declaration. This is a correct declaration utilizing the Servlet 3.0 extension mechanism. Also note that `<servlet-mapping>` is used to define the base resource URI.

When running in a Servlet 2.x then instead it is necessary to declare the Jersey specific servlet and pass the Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] implementation class name as one of the servlet's `init-param` entries:

Example 6.6. Deployment of your application using Jersey specific servlet

```
<web-app>
                <servlet>
                <servlet-name>Jersey Web Application</servlet-name>
                <servlet-class>com.sun.jersey.spi.container.servlet.ServletCon
                <init-param>
                <param-name>javax.ws.rs.Application</param-name>
                <param-value>org.foo.rest.MyApplication</param-value>
                </init-param>
                ...
                </servlet>
                ...
                </web-app>
```

Alternatively a simpler approach is to let Jersey choose the `PackagesResourceConfig` implementation automatically by declaring the packages as follows:

Example 6.7. Using Jersey specific servlet without an application model instance

```
<web-app>
                <servlet>
                <servlet-name>Jersey Web Application</servlet-name>
                <servlet-class>com.sun.jersey.spi.container.servlet.ServletCon
                <init-param>
                <param-name>com.sun.jersey.config.property.packages</param-nam
                <param-value>org.foo.rest;org.bar.rest</param-value>
                </init-param>
                ...
                </servlet>
                ...
                </web-app>
```

JAX-RS also provides the ability to obtain a container specific artifact from an Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] instance. For example, Jersey supports using Grizzly [http://grizzly.java.net/] as follows:

```
SelectorThread st = RuntimeDelegate.createEndpoint(new MyApplication(), SelectorTh
```

Jersey also provides Grizzly [http://grizzly.java.net/] helper classes to deploy the ServletThread instance at a base URL for in-process deployment.

The Jersey samples provide many examples of Servlet-based and Grizzly-in-process-based deployments.

Chapter 7. Client API

This section introduces the JAX-RS Client API, which is a high-level Java based API for interoperating with RESTful Web services. It makes it very easy to interoperate with RESTful Web services and enables a developer to concisely and efficiently implement a reusable client-side solution that leverages existing and well established client-side HTTP implementations.

The client API can be utilized to interoperate with any RESTful Web service, implemented using one of many frameworks, and is not restricted to services implemented using JAX-RS. However, developers familiar with JAX-RS should find the client API complementary to their services, especially if the client API is utilized by those services themselves, or to test those services.

The goals of the client API are threefold:

1. Encapsulate a key constraint of the REST architectural style, namely the Uniform Interface Constraint and associated data elements, as client-side Java artifacts;
2. Make it as easy to interoperate with RESTful Web services as the JAX-RS server-side API makes it easy to build RESTful Web services; and
3. Share common concepts of the JAX-RS API between the server and the client side.

The Client API supports a pluggable architecture to enable the use of different underlying HTTP client implementations. Several such implementations are supported by Jersey. To name a few we have a client connectors for `Http(s)URLConnection` classes supplied with the JDK; and the Grizzly client.

7.1. Uniform Interface Constraint

The uniform interface constraint bounds the architecture of RESTful Web services so that a client, such as a browser, can utilize the same interface to communicate with any service. This is a very powerful concept in software engineering that makes Web-based search engines and service mash-ups possible. It induces properties such as:

1. simplicity, the architecture is easier to understand and maintain; and
2. modifiability or loose coupling, clients and services can evolve over time perhaps in new and unexpected ways, while retaining backwards compatibility.

Further constraints are required:

1. every resource is identified by a URI;
2. a client interacts with the resource via HTTP requests and responses using a fixed set of HTTP methods;
3. one or more representations can be retured and are identified by media types; and
4. the contents of which can link to further resources.

The above process repeated over and again should be familiar to anyone who has used a browser to fill in HTML forms and follow links. That same process is applicable to non-browser based clients.

Many existing Java-based client APIs, such as the Apache HTTP client API or `java.net.HttpURLConnection` supplied with the JDK place too much focus on the Client-Server constraint for the exchanges of request and responses rather than a resource, identified by a URI, and the use of a fixed set of HTTP methods.

A resource in the Jersey client API is an instance of the Java class WebResource [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/WebResource.html], and encapsulates a URI. The fixed set of HTTP methods are methods on `WebResource` or if using the builder pattern (more on this later) are the last methods to be called when invoking an HTTP method on a resource. The representations are Java types, instances of which, may contain links that new instances of `WebResource` may be created from.

7.2. Ease of use and reusing JAX-RS artifacts

Since a resource is represented as a Java type it makes it easy to configure, pass around and inject in ways that is not so intuitive or possible with other client-side APIs.

The Jersey Client API reuses many aspects of the JAX-RS and the Jersey implementation such as:

1. URI building using UriBuilder [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/UriBuilder.html] and UriTemplate [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/uri/UriTemplate.html] to safely build URIs;

2. Support for Java types of representations such as `byte[]`, `String`, `InputStream`, `File`, `DataSource` and JAXB beans in addition to Jersey specific features such as JSON [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/json/package-summary.html] support and MIME Multipart [http://jersey.java.net/nonav/apidocs/2.0/contribs/jersey-multipart/index.html] support.

3. Using the builder pattern to make it easier to construct requests.

Some APIs, like the Apache HTTP client or java.net.HttpURLConnection [http://docs.oracle.com/javase/1.5.0/docs/api/java/net/HttpURLConnection.html], can be rather hard to use and/or require too much code to do something relatively simple.

This is why the Jersey Client API provides support for wrapping HttpURLConnection and the Apache HTTP client. Thus it is possible to get the benefits of the established implementations and features while getting the ease of use benefit.

It is not intuitive to send a POST request with form parameters and receive a response as a JAXB object with such an API. For example with the Jersey API this is very easy:

Example 7.1. POST request with form parameters

```
Form f = new Form();
                    f.add("x", "foo");
                    f.add("y", "bar");

                    Client c = Client.create();
                    WebResource r = c.resource("http://localhost:8080/form");

                    JAXBBean bean = r.
                    type(MediaType.APPLICATION_FORM_URLENCODED_TYPE)
                    .accept(MediaType.APPLICATION_JSON_TYPE)
                    .post(JAXBBean.class, f);

```

In the above code a Form [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/representation/Form.html] is created with two parameters, a new WebResource [http://jersey.java.net/

nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/WebResource.html] instance is created from a Client [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/Client.html] then the `Form` instance is`POST`ed to the resource, identified with the form media type, and the response is requested as an instance of a JAXB bean with an acceptable media type identifying the Java Script Object Notation (JSON) format. The Jersey client API manages the serialization of the `Form` instance to produce the request and de-serialization of the response to consume as an instance of a JAXB bean.

If the code above was written using `HttpURLConnection` then the developer would have to write code to serialize the form sent in the POST request and de-serialize the response to the JAXB bean. In addition further code would have to be written to make it easy to reuse the same resource “http://localhost:8080/form” that is encapsulated in the `WebResource` type.

7.3. Getting started with the Jersey client

Refer to the dependencies chapter [http://jersey.java.net/nonav/documentation/2.0/chapter_deps.html#chapter_deps], and specifically the Core client [http://jersey.java.net/nonav/documentation/2.0/chapter_deps.html#core_client] section, for details on the dependencies when using the Jersey client with Maven and Ant.

Refer to the Java API documentation [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/package-summary.html] for details on the Jersey client API packages and classes.

Refer to the Java API Apache HTTP client documentation [http://jersey.java.net/nonav/apidocs/2.0/contribs/jersey-apache-client/index.html] for details on how to use the Jersey client API with the Apache HTTP client.

7.4. Overview of the API

To utilize the client API it is first necessary to create an instance of a Client [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/Client.html], for example:

```
Client c = Client.create();
```

7.4.1. Configuring a Client and WebResource

The client instance can then be configured by setting properties on the map returned from the `getProperties` methods or by calling the specific setter methods, for example the following configures the client to perform automatic redirection for appropriate responses:

```
c.getProperties().put(
                    ClientConfig.PROPERTY_FOLLOW_REDIRECTS, true);
```

which is equivalent to the following:

```
c.setFollowRedirects(true);
```

Alternatively it is possible to create a `Client` instance using a ClientConfig [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/config/ClientConfig.html] object for example:

```
ClientConfig cc = new DefaultClientConfig();
                    cc.getProperties().put(
                    ClientConfig.PROPERTY_FOLLOW_REDIRECTS, true);
                    Client c = Client.create(cc);
```

Once a client instance is created and configured it is then possible to obtain a WebResource [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/WebResource.html] instance, which will inherit the configuration declared on the client instance. For example, the following creates a reference to a Web resource with the URI “http://localhost:8080/xyz”:

```
WebResource r = c.resource("http://localhost:8080/xyz");
```

and redirection will be configured for responses to requests invoked on the Web resource.

`Client` instances are expensive resources. It is recommended a configured instance is reused for the creation of Web resources. The creation of Web resources, the building of requests and receiving of responses are guaranteed to be thread safe. Thus a `Client` instance and `WebResource` instances may be shared between multiple threads.

In the above cases a `WebResource` instance will utilize `HttpUrlConnection` or `HttpsUrlConnection`, if the URI scheme of the `WebResource` is “http” or “https” respectively.

7.4.2. Building a request

Requests to a Web resource are built using the builder pattern (see RequestBuilder [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/RequestBuilder.html]) where the terminating method corresponds to an HTTP method (see UniformInterface [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/UniformInterface.html]). For example,

```
String response = r.accept(
                    MediaType.APPLICATION_JSON_TYPE,
                    MediaType.APPLICATION_XML_TYPE).
                    header("X-FOO", "BAR").
                    get(String.class);
```

The above sends a GET request with an `Accept` header of `application/json`, `application/xml` and a non-standard header `X-FOO` of `BAR`.

If the request has a request entity (or representation) then an instance of a Java type can be declared in the terminating HTTP method, for `PUT`, `POST` and `DELETE` requests. For example, the following sends a POST request:

```
String request = "content";
                    String response = r.accept(
                    MediaType.APPLICATION_JSON_TYPE,
                    MediaType.APPLICATION_XML_TYPE).
                    header("X-FOO", "BAR").
                    post(String.class, request);
```

where the String "content" will be serialized as the request entity (see the section "Java instances and types for representations" section for further details on the supported Java types). The `Content-Type` of the request entity may be declared using the `type` builder method as follows:

```
String response = r.accept(
                    MediaType.APPLICATION_JSON_TYPE,
                    MediaType.APPLICATION_XML_TYPE).
                    header("X-FOO", "BAR").
                    type(MediaType.TEXT_PLAIN_TYPE).
                    post(String.class, request);
```

or alternatively the request entity and type may be declared using the entity method as follows:

```
String response = r.accept(
                    MediaType.APPLICATION_JSON_TYPE,
                    MediaType.APPLICATION_XML_TYPE).
                    header("X-FOO", "BAR").
                    entity(request, MediaType.TEXT_PLAIN_TYPE).
                    post(String.class);
```

7.4.3. Receiving a response

If the response has a entity (or representation) then the Java type of the instance required is declared in the terminating HTTP method. In the above examples a response entity is expected and an instance of `String` is requested. The response entity will be de-serialized to a String instance.

If response meta-data is required then the Java type ClientResponse [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/ClientResponse.html] can be declared from which the response status, headers and entity may be obtained. For example, the following gets both the entity tag and response entity from the response:

```
ClientResponse response = r.get(ClientResponse.class);
                    EntityTag e = response.getEntityTag();
                    String entity = response.getEntity(String.class);
```

If the `ClientResponse` type is not utilized and the response status is greater than or equal to 300 then the runtime exception UniformInterfaceException [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/UniformInterfaceException.html] is thrown. This exception may be caught and the `ClientResponse` obtained as follows:

```
try {
                    String entity = r.get(String.class);
                    } catch (UniformInterfaceException ue) {
                    ClientResponse response = ue.getResponse();
                    }
```

7.4.4. Creating new WebResources from a WebResource

A new WebResource [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/WebResource.html] can be created from an existing `WebResource` by building from the latter's URI. Thus it is possible to build the request URI before building the request. For example, the following appends a new path segment and adds some query parameters:

```
WebResource r = c.resource("http://localhost:8080/xyz");

                    MultivaluedMap<String, String> params = MultivaluedMapImpl();
                    params.add("foo", "x");
                    params.add("bar", "y");

                    String response = r.path("abc").
                    queryParams(params).
                    get(String.class);
```

that results in a GET request to the URI "http://localhost:8080/xyz/abc?foo=x&bar=y".

7.4.5. Java instances and types for representations

All the Java types for representations supported by the Jersey server side for requests and responses are also supported on the client side. This includes the standard Java types as specified by JAX-RS in section 4.2.4 [http://jsr311.java.net/nonav/releases/1.0/spec/index.html] in addition to JSON, Atom and Multipart MIME as supported by Jersey.

To process a response entity (or representation) as a stream of bytes use InputStream as follows:

```
InputStream in = r.get(InputStream.class);
                    // Read from the stream
                    in.close();
```

Note that it is important to close the stream after processing so that resources are freed up.

To `POST` a file use `File` as follows:

```
File f = ...
                    String response = r.post(String.class, f);
```

Refer to the JAXB sample [https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=jaxb&v=2.0&c=project&e=zip] to see how JAXB with XML and JSON can be utilized with the client API (more specifically, see the unit tests).

7.5. Adding support for new representations

The support for new application-defined representations as Java types requires the implementation of the same provider-based interfaces as for the server side JAX-RS API, namely MessageBodyReader [http://jsr311.java.net/nonav/javadoc/javax/ws/rs/ext/MessageBodyReader.html] and MessageBodyWriter [http://jsr311.java.net/nonav/javadoc/javax/ws/rs/ext/MessageBodyWriter.html], respectively, for request and response entities (or inbound and outbound representations). Refer to the entity provider [https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=entity-provider&v=2.0&c=project&e=zip] sample for such implementations utilized on the server side.

Classes or implementations of the provider-based interfaces need to be registered with a `ClientConfig` and passed to the `Client` for creation. The following registers a provider class `MyReader` which will be instantiated by Jersey:

```
ClientConfig cc = new DefaultClientConfig();
                cc.getClasses().add(MyReader.class);
                Client c = Client.create(cc);
```

The following registers an instance or singleton of MyReader:

```
ClientConfig cc = new DefaultClientConfig();
                MyReader reader = ...
                cc.getSingletons().add(reader);
                Client c = Client.create(cc);
```

7.6. Using filters

Filtering requests and responses can provide useful functionality that is hidden from the application layer of building and sending requests, and processing responses. Filters can read/modify the request URI, headers and entity or read/modify the response status, headers and entity.

The `Client` and `WebResource` classes extend from Filterable [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/filter/Filterable.html] and that enables the addition of ClientFilter [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/filter/ClientFilter.html] instances. A `WebResource` will inherit filters from its creator, which can be a `Client` or another`WebResource`. Additional filters can be added to a `WebResource` after it has been created. For requests, filters are applied in reverse order, starting with the `WebResource` filters and then moving to the inherited filters. For responses, filters are applied in order, starting with inherited filters and followed by the filters added to the`WebResource`. All filters are applied in the order in which they were added. For instance, in the following example the `Client` has two filters added, `filter1` and`filter2`, in that order, and the `WebResource` has one filter added,`filter3`:

```
ClientFilter filter1 = ...
                ClientFilter filter2 = ...
                Client c = Client.create();
                c.addFilter(filter1);
                c.addFilter(filter2);

                ClientFilter filter3 = ...
                WebResource r = c.resource(...);
                r.addFilter(filter3);
```

After a request has been built the request is filtered by`filter3`, `filter2` and `filter1` in that order. After the response has been received the response is filtered by`filter1`, `filter2` and `filter3` in that order, before the response is returned.

Filters are implemented using the “russian doll” stack-based pattern where a filter is responsible for calling the next filter in the ordered list of filters (or the next filter in the “chain” of filters). The basic template for a filter is as follows:

```
class AppClientFilter extends ClientFilter {
                public ClientResponse handle(ClientRequest cr) {
                // Modify the request
                ClientRequest mcr = modifyRequest(cr);
                // Call the next filter
                ClientResponse resp = getNext().handle(mcr);
                // Modify the response
                return modifyResponse(resp);
                }
                }
```

The filter modifies the request (if required) by creating a new ClientRequest [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/ClientRequest.html] or modifying the state of the passed `ClientRequest` before calling the next filter. The call to the next request will return the response, a`ClientResponse`. The filter modifies the response (if required) by creating a new `ClientResponse` or modifying the state of the returned`ClientResponse`. Then the filter returns the modified response. Filters are re-entrant and may be called by multiple threads performing requests and processing responses.

7.6.1. Supported filters

The Jersey Client API currently supports two filters:

1. A GZIP content encoding filter, GZIPContentEncodingFilter [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/filter/GZIPContentEncodingFilter.html]. If this filter is added then a request entity is compressed with the `Content-Encoding` of`gzip`, and a response entity if compressed with a `Content-Encoding` of `gzip` is decompressed. The filter declares an `Accept-Encoding` of`gzip`.

2. A logging filter, LoggingFilter [http://jersey.java.net/nonav/apidocs/2.0/jersey/com/sun/jersey/api/client/filter/LoggingFilter.html]. If this filter is added then the request and response headers as well as the entities are logged to a declared output stream if present, or to `System.out` if not. Often this filter will be placed at the end of the ordered list of filters to log the request before it is sent and the response after it is received.

The filters above are good examples that show how to modify or read request and response entities. Refer to the source code [https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey&a=jersey-client&v=2.0&e=jar] of the Jersey client for more details.

7.7. Testing services

The Jersey client API was originally developed to aid the testing of the Jersey server-side, primarily to make it easier to write functional tests in conjunction with the JUnit framework for execution and reporting. It is used extensively and there are currently over 1000 tests.

Embedded servers, Grizzly and a special in-memory server, are utilized to deploy the test-based services. Many of the Jersey samples contain tests that utilize the client API to server both for testing and examples of how to use the API. The samples utilize Grizzly or embedded Glassfish to deploy the services.

The following code snippets are presented from the single unit test `HelloWorldWebAppTest` of the helloworld-webapp [https://maven.java.net/service/local/artifact/maven/redirect?r=releases&g=com.sun.jersey.samples&a=helloworld-webapp&v=2.0&c=project&e=zip] sample. The `setUp` method, called before a test is executed, creates an instance of the Glassfish server, deploys the application, and a `WebResource` instance that references the base resource:

```
@Override
                protected void setUp() throws Exception {
                super.setUp();

                // Start Glassfish
                glassfish = new GlassFish(BASE_URI.getPort());
                // Deploy Glassfish referencing the web.xml
                ScatteredWar war = new ScatteredWar(
                BASE_URI.getRawPath(), new File("src/main/webapp"),
                new File("src/main/webapp/WEB-INF/web.xml"),
                Collections.singleton(
                new File("target/classes").
                toURI().toURL()));
                glassfish.deploy(war);

                Client c = Client.create();
                r = c.resource(BASE_URI);
                }
```

The `tearDown` method, called after a test is executed, stops the Glassfish server.

```
@Override
                protected void tearDown() throws Exception {
                super.tearDown();
                glassfish.stop();
                }
```

The `testHelloWorld` method tests that the response to a `GET` request to the Web resource returns “Hello World”:

```
public void testHelloWorld() throws Exception {
                String responseMsg = r.path("helloworld").
                get(String.class);
                assertEquals("Hello World", responseMsg);
                }
```

Note the use of the `path` method on the `WebResource` to build from the base`WebResource`.

7.8. Security with Http(s)URLConnection

7.8.1. With Http(s)URLConnection

The support for security, specifically HTTP authentication and/or cookie management with `Http(s)URLConnection` is limited due to constraints in the API. There are currently no specific features or properties on the `Client` class that can be set to support HTTP authentication. However, since the client API, by default, utilizes `HttpURLConnection` or`HttpsURLConnection`, it is possible to configure system-wide security settings (which is obviously not sufficient for multiple client configurations).

For HTTP authentication the `java.net.Authenticator` can be extended and statically registered. Refer to the Http authentication [http://docs.oracle.com/javase/6/docs/technotes/guides/net/http-auth.html] document for more details. For cookie management the `java.net.CookieHandler` can be extended and statically registered. Refer to the Cookie Management [http://docs.oracle.com/javase/6/docs/technotes/guides/net/http-cookie.html] document for more details.

To utilize HTTP with SSL it is necessary to utilize the “https” scheme. For certificate-based authentication see the class HTTPSProperties [http://jersey.java.net/nonav/apidocs/latest/jersey/com/sun/jersey/client/

urlconnection/HTTPSProperties.html] for how to set `javax.net.ssl.HostnameVerifier` and `javax.net.ssl.SSLContext`.

7.8.2. With Apache HTTP client

The support for HTTP authentication and cookies is much better with the Apache HTTP client than with`HttpURLConnection`. See the Java documentation for the package com.sun.jersey.client.apache [http://jersey.java.net/nonav/apidocs/2.0/contribs/jersey-apache-client/com/sun/jersey/client/apache/package-summary.html], ApacheHttpClientState [http://jersey.java.net/nonav/apidocs/2.0/contribs/jersey-apache-client/com/sun/jersey/client/apache/config/ApacheHttpClientState.html] and ApacheHttpClientConfig [http://jersey.java.net/nonav/apidocs/2.0/contribs/jersey-apache-client/com/sun/jersey/client/apache/config/ApacheHttpClientConfig.html] for more details.

Chapter 8. Filters and Interceptors

8.1. Introduction

This chapter describes filters, interceptors and their configuration. Filters and interceptors can be used on both sides, on the client and the server side. Filters can modify inbound and outbound requests and responses including modification of headers, entity and other request/response parameters. Interceptors are used primarily for modification of entity input and output streams. You can use interceptors for example to zip and unzip output and input entity streams.

8.2. Filters

Filters can be used when you want to modify any request or response parameters like headers. For example you would like to add a response header "X-Powered-By" to each generated response. Instead of adding this header in each resource method you would use a response filter to add this header.

There are filters on the server side and the client side.

Server filters:

ContainerRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerRequestFilter.html]
ContainerResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerResponseFilter.html]
Client filters:

ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]
ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]

8.2.1. Server filters

The following example shows a simple container response filter adding a header to each response.

Example 8.1. Container response filter

```
import java.io.IOException;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.core.Response;

public class PoweredByResponseFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext, ContainerRespon
        throws IOException {

            responseContext.getHeaders().add("X-Powered-By", "Jersey :-)");
    }
}
```

In the example above the `PoweredByResponseFilter` always adds a header "X-Powered-By" to the response. The filter must inherit from the ContainerResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/ContainerResponseFilter.html] and must be registered as a provider. The filter will be executed for every response which is in most cases after the resource method is executed. Response filters are executed even if the resource method is not run, for example when the resource method is not found and 404 "Not found" response code is returned by the Jersey runtime. In this case the filter will be executed and will process the 404 response.

The `filter()` method has two arguments, the container request and container response. The ContainerRequestContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ContainerRequestContext.html] is accessible only for read only purposes as the filter is executed already in response phase. The modifications can be done in the ContainerResponseContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ContainerResponseContext.html].

The following example shows the usage of a request filter.

Example 8.2. Container request filter

```
import java.io.IOException;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;

public class AuthorizationRequestFilter implements ContainerRequestFilter {

    @Override
    public void filter(ContainerRequestContext requestContext)
                    throws IOException {

        final SecurityContext securityContext =
                    requestContext.getSecurityContext();
        if (securityContext == null ||
                    !securityContext.isUserInRole("privileged")) {

                requestContext.abortWith(Response
                    .status(Response.Status.UNAUTHORIZED)
                    .entity("User cannot access the resource.")
                    .build());
        }
    }
}
```

The request filter is similar to the response filter but does not have access to the ContainerResponseContext as no response is accessible yet. Response filter inherits from ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]. Request filter is executed before the resource method is run and before the response is created. The filter has possibility to manipulate the request parameters including request headers or entity.

The `AuthorizationRequestFilter` in the example checks whether the authenticated user is in the privileged role. If it is not then the request is *aborted* by calling `ContainerRequestContext.abortWith(Response response)` method. The method is intended to be called from the request filter in situation when the request should not be processed further in the standard processing chain. When the `filter` method is finished the response passed as a parameter to the `abortWith` method is used to respond to the request. Response filters, if any are registered, will be executed and will have possibility to process the aborted response.

8.2.1.1. Pre-matching and post-matching filters

All the request filters shown above was implemented as post-matching filters. It means that the filters would be applied only after a suitable resource method has been selected to process the actual request i.e. after request matching happens. Request matching is the process of finding a resource method that should be executed based on the request path and other request parameters. Since post-matching request filters are invoked when a particular resource method has already been selected, such filters can not influence the resoure method matching process.

To overcome the above described limitation, there is a possibility to mark a server request filter as a *pre-matching* filter, i.e. to annotate the filter class with the @PreMatching [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Prematching.html] annotation. Pre-matching filters are request filters that are executed before the request matching is started. Thanks to this, pre-matching request filters have the possibility to influence which method will be matched. Such a pre-matching request filter example is shown here:

Example 8.3. Pre-matching request filter

```
...
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
...

@PreMatching
public class PreMatchingFilter implements ContainerRequestFilter {

    @Override
    public void filter(ContainerRequestContext requestContext)
                        throws IOException {
        // change all PUT methods to POST
        if (requestContext.getMethod().equals("PUT")) {
            requestContext.setMethod("POST");
        }
    }
}
```

The `PreMatchingFilter` is a simple pre-matching filter which changes all PUT HTTP methods to POST. This might be useful when you want to always handle these PUT and POST HTTP methods with the same Java code. After the `PreMatchingFilter` has been invoked, the rest of the request processing will behave as if the POST HTTP method was originally used. You cannot do this in post-matching filters (standard filters without `@PreMatching` annotation) as the resource method is already matched (selected). An attempt to tweak the original HTTP method in a post-matching filter would cause an `IllegalArgumentException`.

As written above, pre-matching filters can fully influence the request matching process, which means you can even modify request URI in a pre-matching filter by invoking the `setRequestUri(URI)` method of `ContainerRequestFilter` so that a different resource would be matched.

Like in post-matching filters you can abort a response in pre-matching filters too.

8.2.2. Client fillers

Client filters are similar to container filters. The response can also be aborted in the ClientRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientRequestFilter.html] which would cause that no request will actually be sent to the server at all. A new response is passed to the `abort` method. This response will be used and delivered as a result of the request invocation. Such a response goes through the client response filters. This is similar to what happens on the server side. The process is shown in the following example:

Example 8.4. Client request filter

```
public class CheckRequestFilter implements ClientRequestFilter {

    @Override
    public void filter(ClientRequestContext requestContext)
                        throws IOException {
        if (requestContext.getHeaders(
                        ).get("Client-Name") == null) {
            requestContext.abortWith(
                        Response.status(Response.Status.BAD_REQUEST)
                .entity("Client-Name header must be defined.")
                        .build());
         }
    }
}
```

The `CheckRequestFilter` validates the outgoing request. It is checked for presence of a `Client-Name` header. If the header is not present the request will be aborted with a made up response with an appropriate code and message in the entity body. This will cause that the original request will not be effectivelly sent to the server but the actual invocation will still end up with a response as if it would be generated by the server side. If there would be any client response filter it would be executed on this response.

To summarize the workflow, for any client request invoked from the client API the client request filters (ClientRequestFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientRequestFilter.html]) are executed that could manipulate the request. If not aborted, the outcoming request is then physically sent over to the server side and once a response is received back from the server the client response filters (ClientResponseFilter [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/ClientResponseFilter.html]) are executed that might again manipulate the returned response. Finally the response is passed back to the code that invoked the request. If the request was aborted in any client request filter then the client/server communication is skipped and the aborted response is used in the response filters.

8.3. Interceptors

Interceptors share a common API for the server and the client side. Whereas filters are primarily intended to manipulate request and response parameters like HTTP headers, URIs and/or HTTP methods, interceptors are intended to manipulate entities, via manipulating entity input/output streams. If you for example need to encode entity body of a client request then you could implement an interceptor to do the work for you.

There are two kinds of interceptors, ReaderInterceptor [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ReaderInterceptor.html] and WriterInterceptor [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/WriterInterceptor.html]. Reader interceptors are used to manipulate inbound entity streams. These are the streams coming from the "wire". So, using a reader interceptor you can manipulate request entity stream on the server side (where an entity is read from the client request) and response entity stream on the client side (where an entity is read from the server response). Writer interceptors are used for cases where entity is written to the "wire" which on the server means when writing out a response entity and on the client side when writing request entity for a request to be sent out to the server. Writer and reader interceptors are executed before message body readers or writers are executed and their primary intention is to wrap the entity streams that will be used in message body reader and writers.

The following example shows a writer interceptor that enables GZIP compression of the whole entity body.

Example 8.5. GZIP writer interceptor

```
public class GZIPWriterInterceptor implements WriterInterceptor {

    @Override
    public void aroundWriteTo(WriterInterceptorContext context)
                    throws IOException, WebApplicationException {
        final OutputStream outputStream = context.getOutputStream();
        context.setOutputStream(new GZIPOutputStream(outputStream));
        context.proceed();
    }
}
```

The interceptor gets a output stream from the WriterInterceptorContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/WriterInterceptorContext.html] and sets a new one which is a GZIP wrapper of the original output stream. After all interceptors are executed the output stream lastly set to the `WriterInterceptorContext` will be used for serialization of the entity. In the example above the entity bytes will be written to the GZIPOutputStream which will compress the stream data and write them to the original output stream. The original stream is always the stream which writes the data to the "wire". When the interceptor is used on the server, the original output stream is the stream into which writes data to the underlying server container stream that sends the response to the client.

The interceptors wrap the streams and they itself work as wrappers. This means that each interceptor is a wrapper of another interceptor and it is responsibility of each interceptor implementation to call the wrapped interceptor. This is achieved by calling the `proceed()` method on the `WriterInterceptorContext`. This method will call the next registered interceptor in the chain, so effectivelly this will call all remaining registered interceptors. Calling `proceed()` from the last interceptor in the chain will call the appropriate message body reader. Therefore every interceptor must call the `proceed()` method otherwise the entity would not be written. The wrapping principle is reflected also in the method name, aroundWriteTo, which says that the method is wrapping the writing of the entity.

The method aroundWriteTo() gets `WriterInterceptorContext` as a parameter. This context contains getters and setters for header parameters, request properties, entity, entity stream and other properties. These are the properties which will be passed to the final `MessageBodyWriter<T>`. Interceptors are allowed to modify all these properties. This could influence writing of an entity by `MessageBodyWriter<T>` and even selection of such a writer. By changing media type (`WriterInterceptorContext`.setMediaType()) the interceptor can cause that different message body writer will be chosen. The interceptor can also completely replace the entity if it is needed. However, for modification of headers, request properties and such, the filters are usually more preferable choice. Interceptors are executed only when there is any entity and when the entity is to be written. So, when you always want to add a new header to a response no matter what, use filters as interceptors might not be executed when no entity is present. Interceptors should modify properties only for entity serialization and deserialization purposes.

Let's now look at an example of a `WriterInterceptor`

Example 8.6. GZIP reader interceptor

```
public class GZIPReaderInterceptor implements ReaderInterceptor {

    @Override
    public Object aroundReadFrom(ReaderInterceptorContext context)
                    throws IOException, WebApplicationException {
        final InputStream originalInputStream = context.getInputStream();
        context.setInputStream(new GZIPInputStream(originalInputStream));
        return context.proceed();
    }
}
```

The `GZIPReaderInterceptor` wraps the original input stream with the `GZIPInputStream`. All further reads from the entity stream will cause that data will be decompressed by this stream. The interceptor method `aroundReadFrom()` must return an entity. The entity is returned from the `proceed` method of the ReaderInterceptorContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/ext/ReaderInterceptorContext.html]. The `proceed` method internally calls the wrapped interceptor which must also return an entity. The `proceed` method invoked from the last interceptor in the chain calls message body reader which deserializes the entity end returns it. Every interceptor can change this entity if there is a need but in the most cases interceptors will just return the entity as returned from the `proceed` method.

As already mentioned above, interceptors should be primarily used to manipulate entity body. Similar to methods exposed by `WriterInterceptorContext` the `ReaderInterceptorContext` introduces a set of methods for modification of request/response properties like HTTP headers, URIs and/or HTTP methods (excluding getters and setters for entity as entity has not been read yet). Again the same rules as for `WriterInterceptor` applies for changing these properties (change only properties in order to influence reading of an entity).

8.4. Filter and interceptor execution order

Let's look closer at the context of execution of filters and interceptors. The following steps describes scenario where a JAX-RS client makes a POST request to the server. The server receives an entity and sends a response back with the same entity. GZIP reader and writer interceptors are registered on the client and the server. Also filters are registered on client and server which change the headers of request and response.

1. Client request invoked: The POST request with attached entity is built on the client and invoked.
2. ClientRequestFilters: The ClientResponseFilters are executed on the client and they manipulate the request headers.
3. Client `WriterInterceptor`: As the request contains an entity, writer interceptor registered on the client is executed before a MessageBodyWriter is executed. It wraps the entity output stream with the GZipOutputStream.
4. Client MessageBodyWriter: message body writer is executed on the client which serializes the entity into the new GZipOutput stream. This stream zips the data and sends it to the "wire".
5. Server: server receives a request. Data of entity is compressed which means that pure read from the entity input stream would return compressed data.
6. Server pre-matching ContainerRequestFilters: ContainerRequestFilters are executed that can manipulate resource method matching process.

7. Server: matching: resource method matching is done.

8. Server: post-matching ContainerRequestFilters: ContainerRequestFilters post matching filters are executed. This include execution of all global filters (without name binding) and filters name-bound to the matched method.

9. Server `ReaderInterceptor`: reader interceptors are executed on the server. The GZIPReaderInterceptor wraps the input stream (the stream from the "wire") into the GZipInputStream and set it to context.

10. Server MessageBodyReader: server message body reader is executed and it deserializes the entity from new GZipInputStream (get from the context). This means the reader will read unzipped data and not the compressed data from the "wire".

11. Server resource method is executed: the deserialized entity object is passed to the matched resource method as a parameter. The method returns this entity as a response entity.

12. Server ContainerResponseFilters are executed: response filters are executed on the server and they manipulate the response headers. This include all global bound filters (without name binding) and all filters name-bound to the resource method.

13. Server `WriterInterceptor`: is executed on the server. It wraps the original output stream with a new GZIPOuptutStream. The original stream is the stream that "goes to the wire" (output stream for response from the underlying server container).

14. Server MessageBodyWriter: message body writer is executed on the server which serializes the entity into the GZIPOutputStream. This stream compresses the data and writes it to the original stream which sends this compressed data back to the client.

15. Client receives the response: the response contains compressed entity data.

16. Client ClientResponseFilters: client response filters are executed and they manipulate the response headers.

17. Client response is returned: the javax.ws.rs.core.Response is returned from the request invocation.

18. Client code calls response.readEntity(): read entity is executed on the client to extract the entity from the response.

19. Client `ReaderInterceptor`: the client reader interceptor is executed when readEntity(Class) is called. The interceptor wraps the entity input stream with GZIPInputStream. This will decompress the data from the original input stream.

20. Client MessageBodyReaders: client message body reader is invoked which reads decompressed data from GZIPInputStream and deserializes the entity.

21. Client: The entity is returned from the readEntity().

It is worth to mention that in the scenario above the reader and writer interceptors are invoked only if the entity is present (it does not make sense to wrap entity stream when no entity will be written). The same behaviour is there for message body readers and writers. As mentioned above, interceptors are executed before the message body reader/writer as a part of their execution and they can wrap the input/output stream before the entity is read/written. There are exceptions when interceptors are not run before message body reader/writers but this is not the case of simple scenario above. This happens for example when the entity is read many times from client response using internal buffering. Then the data are intercepted only once and kept 'decoded' in the buffer.

8.5. Name binding

Filters and interceptors can be *name-bound*. Name binding is a concept that allows to say to a JAX-RS runtime that a specific filter or interceptor will be executed only for a specific resource method. When a filter or an interceptor is limited only to a specific resource method we say that it is *name-bound*. Filters and interceptors that do not have such a limitation are called *global*.

Filter or interceptor can be assigned to a resource method using the @NameBinding [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/NameBinding.html] annotation. The annotation is used as meta annotation for other user implemented annotations that are applied to a providers and resource methods. See the following example:

Example 8.7. `@NameBinding` example

```
...
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.util.zip.GZIPInputStream;

import javax.ws.rs.GET;
import javax.ws.rs.NameBinding;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
...


// @Compress annotation is the name binding annotation
@NameBinding
@Retention(RetentionPolicy.RUNTIME)
public @interface Compress {}


@Path("helloworld")
public class HelloWorldResource {

    @GET
    @Produces("text/plain")
    public String getHello() {
        return "Hello World!";
    }

    @GET
    @Path("too-much-data")
    @Compress
    public String getVeryLongString() {
        String str = ... // very long string
        return str;
    }
}

// interceptor will be executed only when resource methods
// annotated with @Compress annotation will be executed
@Compress
public class GZIPWriterInterceptor implements WriterInterceptor {
    @Override
    public void aroundWriteTo(WriterInterceptorContext context)
                    throws IOException, WebApplicationException {
        final OutputStream outputStream = context.getOutputStream();
        context.setOutputStream(new GZIPOutputStream(outputStream));
        context.proceed();
    }
}
```

The example above defines a new `@Compress` annotation which is a name binding annotation as it is annotated with `@NameBinding`. The `@Compress` is applied on the resource method `getVeryLongString()` and on the interceptor `GZIPWriterInterceptor`. The interceptor will be executed only if any resource method with such a annotation will be executed. In our example case the interceptor will be executed only for the `getVeryLongString()` method. The interceptor will not be executed for method `getHello()`. In this example the reason is probably clear. We would like to compress only long data and we do not need to compress the short response of "Hello World!".

Name binding can be applied on a resource class. In the example `HelloWorldResource` would be annotated with `@Compress`. This would mean that all resource methods will use compression in this case.

There might be many name binding annotations defined in an application. When any provider (filter or interceptor) is annotated with more than one name binding annotation, then it will be executed for resource methods which contain ALL these annotations. So, for example if our interceptor would be annotated with another name binding annotation @GZIP then the resource method would need to have both annotations attached, @Compress and @GZIP, otherwise the interceptor would not be executed. Based on the previous paragraph we can even use the combination when the resource method `getVeryLongString()` would be annotated with @Compress and resource class `HelloWorldResource` would be annotated from with @GZIP. This would also trigger the interceptor as annotations of resource methods are aggregated from resource method and from resource class. But this is probably just an edge case which will not be used so often.

Note that *global filters are executed always*, so even for resource methods which have any name binding annotations.

8.6. Dynamic binding

Dynamic binding is a way how to assign filters and interceptors to the resource methods in a dynamic manner. Name binding from the previous chapter uses a static approach and changes to binding require source code change and recompilation. With dynamic binding you can implement code which defines bindings during the application initialization time. The following example shows how to implement dynamic binding.

Example 8.8. Dynamic binding example

```
...
import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.container.DynamicFeature;
...

@Path("helloworld")
public class HelloWorldResource {

    @GET
    @Produces("text/plain")
    public String getHello() {
        return "Hello World!";
    }

    @GET
    @Path("too-much-data")
    public String getVeryLongString() {
        String str = ... // very long string
        return str;
    }
}

// This dynamic binding provider registers GZIPWriterInterceptor
// only for HelloWorldResource and methods that contain
// "VeryLongString" in their name. It will be executed during
// application initialization phase.
public class CompressionDynamicBinding implements DynamicFeature {

    @Override
    public void configure(ResourceInfo resourceInfo, FeatureContext context) {
        if (HelloWorldResource.class.equals(resourceInfo.getResourceClass())
                && resourceInfo.getResourceMethod()
                    .getName().contains("VeryLongString")) {
            context.register(GZIPWriterInterceptor.class);
        }
    }
}
```

The example contains one `HelloWorldResource` which is known from the previous name binding example. The difference is in the `getVeryLongString` method, which now does not define the `@Compress` name binding annotations. The binding is done using the provider which implements DynamicFeature [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/container/DynamicFeature.html] interface. The interface defines one `configure` method with two arguments, `ResourceInfo` and `FeatureContext`. `ResourceInfo` contains information about the resource and method to which the binding can be done. The `configure` method will be executed once for each resource method that is defined in the application. In the example above the provider will be executed twice, once for the `getHello()` method and once for `getVeryLongString()` (once the resourceInfo will contain information about getHello() method and once it will point to getVeryLongString()). If a dynamic binding provider wants to register any provider for the actual resource method it will do that using provided `FeatureContext` which extends JAX-RS `Configurable` API. All methods for registration of filter or interceptor classes or instances can be used. Such dynamically registered filters or interceptors will be bound only to the actual resource method. In the example above the `GZIPWriterInterceptor` will be bound only to the method `getVeryLongString()` which will cause that data will be compressed only for this method and not for the method `getHello()`. The code of `GZIPWriterInterceptor` is in the examples above.

Note that filters and interceptors registered using dynamic binding are only additional filters run for the resource method. If there are any name bound providers or global providers they will still be executed.

8.7. Priorities

In case you register more filters and interceptors you might want to define an exact order in which they should be invoked. The order can be controlled by the `@Priority` annotation defined by the `javax.annotation.Priority` class. The annotation accepts an integer parameter of priority. Providers used in request processing (ContainerRequestFilter, ClientRequestFilter, ReaderInterceptors) are sorted based on the priority in an ascending manner. So, a request filter with priority defined with `@Priority(1000)` will be executed before another request filter with priority defined as `@Priority(2000)`. Providers used during response processing (ContainerResponseFilter, ClientResponseFilter, WriterIntercepors) are executed in the reverse order (using descending manner), so a provider with the priority defined with `@Priority(2000)` will be executed before another provider with priority defined with `@Priority(1000)`.

It's a good practice to assign a priority to filters and interceptors. Use Priorities [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Priorities.html] class which defines standardized priorities in JAX-RS for different usages, rather than inventing your own priorities. So, when you for example write an authentication filter you would assign a priority 1000 which is the value of `Priorities.AUTHENTICATION`. The following example shows the filter from the beginning of this chapter with a priority assigned.

Example 8.9. Priorities example

```
...
import javax.annotation.Priority;
import javax.ws.rs.Priorities;
...

@Priority(Priorities.HEADER_DECORATOR)
public class ResponseFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
                    ContainerResponseContext responseContext)
                    throws IOException {

        responseContext.getHeaders().add("X-Powered-By", "Jersey :-)");
    }
}
```

As this is a response filter and response filters are executed in the reverse order, any other filter with priority lower than 3000 (`Priorities.HEADER_DECORATOR` is 3000) will be executed after this filter. So, for example `AUTHENTICATION` filter (priority 1000) would be run after this filter.

Chapter 9. Message Body Workers

TODO: Describe message body readers and writers (how to write custom ones)

Chapter 10. Asynchronous Services and Clients

TODO: Describe asynchronous services, ChunkedResponse and async client

Chapter 11. Programmatic API for Building Resources

11.1. Introduction

A standard approach of developing JAX-RS application is to implement resource classes annotated with @Path [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/Path.html] with resource methods annotated with HTTP method annotations like @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html] or @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html] and implement needed functionality. This approach is described in the chapter JAX-RS Application, Resources and Sub-Resources [jaxrs-resources.html]. Besides this standard JAX-RS approach, Jersey offers an API for constructing resources programmatically.

Imagine a situation where a deployed JAX-RS application consists of many resource classes. These resources implement standard HTTP methods like @GET [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/GET.html], @POST [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/POST.html], @DELETE [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/DELETE.html]. In some situations it would be useful to add an @OPTIONS [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/OPTIONS.html] method which would return some kind of meta data about the deployed resource. Ideally, you would want to expose the functionality as an additional feature and you want to decide at the deploy time whether you want to add your custom `OPTIONS` method. However, when custom `OPTIONS` method are not enabled you would like to be `OPTIONS` requests handled in the standard way by JAX-RS runtime. To achieve this you would need to modify the code to add or remove custom `OPTIONS` methods before deployment. Another way would be to use programmatic API to build resource according to your needs.

Another use case of programmatic resource builder API is when you build any generic RESTful interface which depends on lot of configuration parameters or for example database structure. Your resource classes would need to have different methods, different structure for every new application deploy. You could use more solutions including approaches where your resource classes would be built using Java byte code manipulation. However, this is exactly the case when you can solve the problem cleanly with the programmatic resource builder API. Let's have a closer look at how the API can be utilized.

11.2. Programmatic Hello World example

Jersey Programmatic API was designed to fully support JAX-RS resource model. In other words, every resource that can be designed using standard JAX-RS approach via annotated resource classes can be also modelled using Jersey programmatic API. Let's try to build simple hello world resource using standard approach first and then using the Jersey programmatic resource builder API.

The following example shows standard JAX-RS "Hello world!" resource class.

Example 11.1. A standard resource class

```

@Path("helloworld")
public class HelloWorldResource {

    @GET
    @Produces("text/plain")
    public String getHello() {
        return "Hello World!";
    }
}

```

This is just a simple resource class with one GET method returning "Hello World!" string that will be serialized as text/plain media type.

Now we will design this simple resource using programmatic API.

Example 11.2. A programmatic resource

```

package org.glassfish.jersey.examples.helloworld;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.core.Application;
import javax.ws.rs.core.Response;
import org.glassfish.jersey.process.Inflector;
import org.glassfish.jersey.server.ResourceConfig;
import org.glassfish.jersey.server.model.Resource;
import org.glassfish.jersey.server.model.ResourceMethod;


public static class MyResourceConfig extends ResourceConfi

    public MyResourceConfig() {
        final Resource.Builder resourceBuilder = Resource.
        resourceBuilder.path("helloworld");

        final ResourceMethod.Builder methodBuilder = resou
        final ResourceMethod.Builder methodBuilder = resou
        methodBuilder.produces(MediaType.TEXT_PLAIN_TYPE)
                .handledBy(new Inflector<ContainerRequestC

            @Override
            public String apply(ContainerRequestContext co
                return "Hello World!";
            }
        });

        final Resource resource = resourceBuilder.build();
        registerResources(resource);
    }
}

```

First, focus on the content of the `MyResourceConfig` constructor in the example. The Jersey programmatic resource model is constructed from `Resource`s that contain `ResourceMethod`s. In the example, a single resource would be constructed from a `Resource` containing one `GET` resource method that returns "Hello World!". The main entry point for building programmatic resources in Jersey is the *`Resource.Builder`* class. `Resource.Builder` contains just a few methods like the `path` method used in the example, which sets the name of the path. Another useful method is a `addMethod(String path)` which adds a new method to the resource builder and returns a resource method builder for the new method. Resource method builder contains methods which set consumed and produced media type, define name bindings, timeout for asynchronous executions, etc. It is always necessary to define a resource method handler (i.e. the code of the resource method that will return "Hello World!"). There are more options how a resource method can be handled. In the example the implementation of `Inflector` is used. The Jersey `Inflector` interface defines a simple contract for transformation of a request into a response. An inflector can either return a Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html] or directly an entity object, the way it is shown in the example. Another option is to setup a Java method handler using `handledBy(Class<?> handlerClass, Method method)` and pass it the chosen `java.lang.reflect.Method` instance together with the enclosing class.

A resource method model construction can be explicitly completed by invoking `build()` on the resource method builder. It is however not necessary to do so as the new resource method model will be built automatically once the parent resource is built. Once a resource model is built, it is registered into the resource config at the last line of the `MyResourceConfig` constructor in the example.

11.2.1. Deployment of programmatic resources

Let's now look at how a programmatic resources are deployed. The easiest way to setup your application as well as register any programmatic resources in Jersey is to use a Jersey `ResourceConfig` utility class, an extension of Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] class. If you deploy your Jersey application into a Servlet container you will need to provide a Application [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Application.html] subclass that will be used for configuration. You may use a `web.xml` where you would define a `org.glassfish.jersey.servlet.ServletContainer` Servlet entry there and specify initial parameter `javax.ws.rs.Application` with the class name of your JAX-RS Application that you wish to deploy. In the example above, this application will be `MyResourceConfig` class. This is the reason why `MyResourceConfig` extends the `ResourceConfig` (which, if you remember extends the `javax.ws.rs.Application`).

The following example shows a fragment of `web.xml` that can be used to deploy the `ResourceConfig` JAX-RS application.

Example 11.3. A programmatic resource

```

                            ...
                            <servlet>
                                <servlet-name>org.glassfish.jersey.examples.hellow
                                <servlet-class>org.glassfish.jersey.servlet.Servle
                                <init-param>
                                    <param-name>javax.ws.rs.Application</param-nam
                                    <param-value>org.glassfish.jersey.examples.hel
                                </init-param>
                                <load-on-startup>1</load-on-startup>
                            </servlet>
                            <servlet-mapping>
                                <servlet-name>org.glassfish.jersey.examples.hellow
                                <url-pattern>/*</url-pattern>
                            </servlet-mapping>
                            ...

```

If you use another deployment options and you have accessible instance of ResourceConfig which you use for configuration, you can register programmatic resources directly by `registerResources()` method called on the ResourceConfig. Please note that the method registerResources() replaces all the previously registered resources.

Because Jersey programmatic API is not a standard JAX-RS feature the `ResourceConfig` must be used to register programmatic resources as shown above. See deployment chapter for more information.

11.3. Additional examples

Example 11.4. A programmatic resource

```

                        final Resource.Builder resourceBuilder = Resource.builder(
                        resourceBuilder.addMethod("OPTIONS")
                            .handledBy(new Inflector<ContainerRequestContext, Resp
                                @Override
                                public Response apply(ContainerRequestContext cont
                                    return Response.ok("This is a response to an O
                                }
                            });
                        final Resource resource = resourceBuilder.build();

```

In the example above the `Resource` is built from a `HelloWorldResource` resource class. The resource model built this way contains a `GET` method producing `'text/plain'` responses with "Hello World!" entity. This is quite important as it allows you to whatever Resource objects based on introspecting existing JAX-RS resources and use builder API to enhance such these standard resources. In the example we used already implemented `HelloWorldResource` resource class and enhanced it by `OPTIONS` method. The `OPTIONS` method is handled by an Inflector which returns Response [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Response.html].

The following sample shows how to define sub-resource methods (methods that contains sub-path).

Example 11.5. A programmatic resource

```

                    final Resource.Builder resourceBuilder = Resource.builder(

                    final Resource.Builder childResource = resourceBuilder.add
                    childResource.addMethod("GET").handledBy(new GetInflector(

                    final Resource resource = resourceBuilder.build();

```

Sub-resource methods are defined using so called *child resource models*. Child resource models (or child resources) are programmatic resources build in the same way as any other programmatic resource but they are registered as a child resource of a parent resource. The child resource in the example is build directly from the parent builder using method `addChildResource(String path)`. However, there is also an option to build a child resource model separately as a standard resource and then add it as a child resource to a selected parent resource. This means that child resource objects can be reused to define child resources in different parent resources (you just build a single child resource and then register it in multiple parent resources). Each child resource groups methods with the same sub-resource path. Note that a child resource cannot have any child resources as there is nothing like sub-sub-resource method concept in JAX-RS. For example if a sub resource method contains more path segments in its path (e.g. "/root/sub/resource/method" where "root" is a path of the resource and "sub/resource/method" is the sub resource method path) then parent resource will have path "root" and child resource will have path "sub/resource/method" (so, there will not be any separate nested sub-resources for "sub", "resource" and "method").

See the javadocs of the resource builder API for more information.

11.4. Model processors

Jersey gives you an option to register so called *model processor providers*. These providers are able to modify or enhance the application resource model during application deployment. This is an advanced feature and will not be needed in most use cases. However, imagine you would like to add `OPTIONS` resource method to each deployed resource as it is described at the top of this chapter. You would want to do it for every programmatic resource that is registered as well as for all standard JAX-RS resources.

To do that, you first need to register a model processor provider in your application, which implements `org.glassfish.jersey.server.model.ModelProcessor` extension contract. An example of a model processor implementation is shown here:

Example 11.6. A programmatic resource

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.core.Application;
import javax.ws.rs.core.Configuration;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;

import org.glassfish.jersey.process.Inflector;
import org.glassfish.jersey.server.model.ModelProcessor;
import org.glassfish.jersey.server.model.Resource;
import org.glassfish.jersey.server.model.ResourceMethod;
import org.glassfish.jersey.server.model.ResourceModel;

@Provider
public static class MyOptionsModelProcessor implements Mod

    @Override
    public ResourceModel processResourceModel(ResourceMode
        // we get the resource model and we want to enhanc
        ResourceModel.Builder newResourceModelBuilder = ne
        for (final Resource resource : resourceModel.getRe
            // for each resource in the resource model we
            final Resource.Builder resourceBuilder = Resou

            // we add a new OPTIONS method to each resourc
            // note that we should check whether the metho
            resourceBuilder.addMethod("OPTIONS")
                .handledBy(new Inflector<ContainerRequestC
                    @Override
                    public String apply(ContainerRequestCo
                        return "On this path the resource
                            + " methods is deployed.";
                    }
                    }).produces(MediaType.TEXT_PLAIN);

            // we add to the model new resource which is a
            // by the OPTIONS method
            newResourceModelBuilder.addResource(resourceBu
        }

        final ResourceModel newResourceModel = newResource
        // and we return new model
        return newResourceModel;
    };

    @Override
    public ResourceModel processSubResource(ResourceModel
        // we just return the original subResourceModel wh
        return subResourceModel;
    }
}
```

The `MyOptionsModelProcessor` from the example is a relatively simple model processor which iterates over all registered resources and for all of them builds a new resource that is equal to the old resource except it is enhanced with a new `OPTIONS` method.

Note that you only need to register such a ModelProcessor as your custom extension provider in the same way as you would register any standard JAX-RS extension provider. During an application deployment, Jersey will look for any registered model processor and execute them. As you can seem, model processors are very powerful as they can do whatever manipulation with the resource model they like. A model processor can even, for example, completely ignore the old resource model and return a new custom resource model with a single "Hello world!" resource, which would result in only the "Hello world!" resource being deployed in your application. Of course, it would not not make much sense to implement such model processor, but the scenario demonstrates how powerful the model processor concept is. A better, real-life use case scenario would, for example, be to always add some custom new resource to each application that might return additional metadata about the deployed application. Or, you might want to filter out particular resources or resource methods, which is another situation where a model processor could be used successfully.

Also note that `processSubResource(ResourceModel subResourceModel, Configuration configuration)` method is executed for each sub resource returned from the sub resource locator. The example is simplified and does not do any manipulation but probably in such a case you would want to enhance all sub resources with a new `OPTIONS` method handlers too.

It is important to remember that any model processor must always return valid resource model. As you might have already noticed, in the example above this important rule is not followed. If any of the resources in the original resource model would already have an `OPTIONS` method handler defined, adding another handler would cause the application fail during the deployment in the resource model validation phase. In order to retain the consistency of the final model, a model processor implementation would have to be more robust than what is shown in the example.

Chapter 12. Support for Common Media Types

12.1. JSON

TODO: Describe support for JSON (various notations/options)

12.2. XML

TODO: Describe support for XML

12.3. Multipart

TODO: Describe support for multipart

Chapter 13. Support for Server-Sent Events

TODO: Describe support for SSE

Chapter 14. Security

Security information is available by obtaining the SecurityContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/SecurityContext.html] using @Context [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/Context.html], which is essentially the equivalent functionality available on the HttpServletRequest [http://docs.oracle.com/javaee/5/api/javax/servlet/http/HttpServletRequest.html].

SecurityContext [http://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/core/SecurityContext.html] can be used in conjunction with sub-resource locators to return different resources if the user principle is included in a certain role. For example, a sub-resource locator could return a different resource if a user is a preferred customer:

Example 14.1. Accessing `SecurityContext`

```
@Path("basket")
public ShoppingBasketResource get(@Context SecurityContext sc) {
    if (sc.isUserInRole("PreferredCustomer") {
       return new PreferredCustomerShoppingBaskestResource();
    } else {
       return new ShoppingBasketResource();
    }
}
```

Chapter 15. WADL Support

TODO: Describe WADL support

Chapter 16. Jersey Test Framework

TODO: Describe Test Framework

Chapter 17. Building and Testing Jersey

17.1. Checking Out the Source

Jersey source code is available in a Git repository you can browse at http://java.net/projects/jersey/sources/code/show.

In case you are not familiar with Git, we recommend reading on of the many "Getting Started with Git" articles you can find on the web. For example this DZone RefCard [http://refcardz.dzone.com/refcardz/getting-started-git].

Before you can clone Jersey repository you have to sign up for a java.net [http://java.net] account. Once you are registered, you have to add an SSH key to your java.net profile - see this article on how to do that: http://java.net/projects/help/pages/ProfileSettings#SSH_Keys_Tab

To clone the Jersey repository you can execute the following command on the command-line (provided you have a command-line Git client installed on your machine):

```
git clone ssh://<your_java_net_id>@java.net/jersey~code
```

Milestones and releases of Jersey are tagged. You can list the tags by executing the standard Git command in the repository directory:

```
git tag -l
```

17.2. Building the Source

Jersey source code requires Java SE 6 or greater. The build is based on Maven. Maven 3 or greater is recommended. Also it is recommended you use the following Maven options when building the workspace (can be set in MAVENT_OPTS environment variable):

```
-Xmx1048m -XX:PermSize=64M -XX:MaxPermSize=128M
```

It is recommended to build all of Jersey after you cloned the source code repository. To do that execute the following commands in the directory where jersey source repository was cloned (typically the directory named "jersey~code"):

```
mvn -Dmaven.test.skip=true clean install
```

This command will build Jersey, but skip the test execution. If you don't want to skip the tests, execute the following instead:

```
mvn clean install
```

Building the whole Jersey project including tests could take significant amount of time.

17.3. Testing

Jersey contains many tests. Unit tests are in the individual Jersey modules, integration and end-to-end tests are in jersey~code/tests directory. You can run tests related to a particular area using the following command:

```
mvn -Dtest=<pattern> test
```

where `pattern` may be a comma separated set of names matching tests.

17.4. Using NetBeans

NetBeans IDE [http://netbeans.org] has excellent maven support. The Jersey maven modules can be loaded, built and tested in NetBeans without any additional NetBeans-specific project files.

Chapter 18. Migrating from Jersey 1.x

This chapter is a migration guide for people switching from Jersey 1.x. Since many of the Jersey 1.x features became part of JAX-RS 2.0 standard which caused changes in the package names, we decided it is a good time to do a more significant incompatible refactoring, which will allow us to introduce some more interesting new features in the future. As the result, there are many incompatiblities between Jersey 1.x and Jersey 2.0. This chapter summarizes how to migrate the concepts found in Jersey 1.x to Jersey/ JAX-RS 2.0 concepts.

18.1. Server API

Jersey 1.x contains number of proprietary server APIs. This section covers migration of application code relying on those APIs.

18.1.1. Injecting custom objects

Jersey 1.x have its own internal dependency injection framework which handles injecting various parameters into field or methods. It also provides a way how to register custom injection provider in Singleton or PerRequest scopes. Jersey 2.x uses HK2 as dependency injection framework and users are also able to register custom classes or instances to be injected in various scopes.

Main difference in Jersey 2.x is that you don't need to create special classes or providers for this task; everything should be achievable using HK2 API. Custom injectables can be registered at ResourceConfig level by adding new HK2 Module or by dynamically adding binding almost anywhere using injected HK2 Services instance.

Jersey 1.x Singleton:

```
ResourceConfig resourceConfig = new DefaultResourceConfig();
                    resourceConfig.getSingletons().add(
                    new SingletonTypeInjectableProvider<Context, SingletonType>(
                    SingletonType.class, new SingletonType()) {});
```

Jersey 1.x PerRequest:

```
ResourceConfig resourceConfig = new DefaultResourceConfig();
                    resourceConfig.getSingletons().add(
                    new PerRequestTypeInjectableProvider<Context, PerRequestType>(
                    @Override
                    public Injectable<PerRequestType> getInjectable(ComponentConte
                    //...
                    }
                    });
```

Jersey 2.0 HK2 Module:

```
public static class MyBinder extends AbstractBinder {

                    @Override
                    protected void configure() {
                    // request scope binding
                    bind(MyInjectablePerRequest.class).to(MyInjectablePerRequest.c
                    // singleton binding
                    bind(MyInjectableSingleton.class).in(Singleton.class);
                    // singleton instance binding
                    bind(new MyInjectableSingleton()).to(MyInjectableSingleton.cla
                    }

                    }

                    // register module to ResourceConfig (can be done also in cons
                    ResourceConfig rc = new ResourceConfig();
                    rc.addClasses(/* ... */);
                    rc.addBinders(new MyBinder());
```

Jersey 2.0 dynamic binding:

```
public static class MyApplication extends Application {

                    @Inject
                    public MyApplication(ServiceLocator serviceLocator) {
                    System.out.println("Registering injectables...");

                    DynamicConfiguration dc = Injections.getConfiguration(serviceL

                    // request scope binding
                    Injections.addBinding(
                    Injections.newBinder(MyInjectablePerRequest.class).to(MyInject
                    dc);

                    // singleton binding
                    Injections.addBinding(
                    Injections.newBinder(MyInjectableSingleton.class).to(MyInjecta
                    dc);

                    // singleton instance binding
                    Injections.addBinding(
                    Injections.newBinder(new MyInjectableSingleton()).to(MyInjecta
                    dc);

                    // request scope binding with specified custom annotation
                    Injections.addBinding(
                    Injections.newBinder(MyInjectablePerRequest.class).to(MyInject
                    .qualifiedBy(new MyAnnotationImpl())
                    .in(RequestScoped.class),
                    dc);

                    // commits changes
                    dc.commit();
                    }

                    @Override
                    public Set<Class<?>> getClasses() {
                    return ...
```

```
                              }
                              }
```

18.1.2. ResourceConfig Reload

In Jersey 1, the reload functionality is based on two interfaces:

1. com.sun.jersey.spi.container.ContainerListener

2. com.sun.jersey.spi.container.ContainerNotifier

Containers, which support the reload functionality implement the `ContainerListener` interface, so that once you get access to the actual container instance, you could call it's `onReload` method and get the container re-load the config. The second interface helps you to obtain the actual container instance reference. An example on how things are wired together follows.

Example 18.1. Jersey 1 reloader implementation

```
public class Reloader implements ContainerNotifier {
                        List<ContainerListener> ls;

                        public Reloader() {
                        ls = new ArrayList<ContainerListener>();
                        }

                        public void addListener(ContainerListener l) {
                        ls.add(l);
                        }

                        public void reload() {
                        for (ContainerListener l : ls) {
                        l.onReload();
                        }
                        }
                        }

```

Example 18.2. Jersey 1 reloader registration

```
Reloader reloader = new Reloader();
                        resourceConfig.getProperties().put(ResourceConfig.PROP

```

In Jersey 2, two interfaces are involved again, but these have been re-designed.

1. org.glassfish.jersey.server.spi.Container

2. org.glassfish.jersey.server.spi.ContainerLifecycleListener

The `Container` interface introduces two `reload` methods, which you can call to get the application re-loaded. One of these methods allows to pass in a new `ResourceConfig` instance. You can register your implementation of ContainerLifecycleListener the same way as any other provider (i.e. either by annotating it by @Provider annotation or adding it to the ResourceConfig directly either using the class (using

ResourceConfig.addClasses()) or registering a particular instance using ResourceConfig.addSingletons() method. An example on how things work in Jersey 2 follows.

Example 18.3. Jersey 2 reloader implementation

```
public class Reloader implements ContainerLifecycleListener {

                        Container container;

                        public void reload(ResourceConfig newConfig) {
                        container.reload(newConfig);
                        }

                        public void reload() {
                        container.reload();
                        }

                        @Override
                        public void onStartup(Container container) {
                        this.container = container;
                        }

                        @Override
                        public void onReload(Container container) {
                        // ignore or do whatever you want after reload has bee
                        }

                        @Override
                        public void onShutdown(Container container) {
                        // ignore or do something after the container has been
                        }
                        }

```

Example 18.4. Jersey 2 reloader registration

```
Reloader reloader = new Reloader();
                        resourceConfig.addSingletons(reloader);

```

18.1.3. MessageBodyReaders and MessageBodyWriters ordering

JAX-RS 2.0 defines new order of MessageBodyWorkers - whole set is sorted by declaration distance, media type and source (custom providers have smaller priority than Jersey provided). JAX-RS 1.x ordering can still be forced by setting parameter MessageProperties.LEGACY_WORKERS_ORDERING ("jersey.config.workers.legacyOrdering") to true in ResourceConfig or ClientConfig properties.

18.2. Client API

JAX-RS 2.0 provides functionality that is equivalent to the Jersey 1.x proprietary client API. Here is a rough mapping between the Jersey 1.x and JAX-RS 2.0 Client API classes:

Table 18.1. Mapping of Jersey 1.x to JAX-RS 2.0 client classes

Jersey 1.x Class	JAX-RS 2.0 Class	Notes
com.sun.jersey.api.client.Client [http://jersey.java.net/nonav/apidocs/latest/jersey/com/sun/jersey/api/client/Client.html]	javax.ws.rs.client.ClientBuilder [http://jax-rs-spec.java.net/nonav/2.0-SNAPSHOT/apidocs/javax/ws/rs/client/ClientBuilder.html]	For the static factory methods and constructors.
	javax.ws.rs.client.Client [http://jax-rs-spec.java.net/nonav/2.0-SNAPSHOT/apidocs/javax/ws/rs/client/Client.html]	For the instance methods.
com.sun.jersey.api.client.WebResource [http://jersey.java.net/nonav/apidocs/latest/jersey/com/sun/jersey/api/client/WebResource.html]	javax.ws.rs.client.WebTarget [http://jax-rs-spec.java.net/nonav/2.0-SNAPSHOT/apidocs/javax/ws/rs/client/WebTarget.html]	
com.sun.jersey.api.client.AsyncWebResource [http://jersey.java.net/nonav/apidocs/latest/jersey/	javax.ws.rs.client.WebTarget [http://jax-rs-spec.java.net/nonav/2.0-SNAPSHOT/	You can access async versions of the async methods by calling WebTarget.request().async()

Jersey 1.x Class	**JAX-RS 2.0 Class**
com/ sun/ jersey/ api/ client/ AsyncWebResource.html]	apidocs/ javax/ ws/ rs/ client/ WebTarget.html]

The following sub-sections show code examples.

18.2.1. Making a simple client request

Jersey 1.x way:

```
Client client = Client.create();
                    WebResource webResource = client.resource(restURL).path("myres
                    String result = webResource.pathParam("param", "value").get(St
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
                    WebTarget target = client.target(restURL).path("myresource/{pa
                    String result = target.pathParam("param", "value").get(String.
```

18.2.2. Registering filters

Jersey 1.x way:

```
Client client = Client.create();
                    WebResource webResource = client.resource(restURL);
                    webResource.addFilter(new HTTPBasicAuthFilter(username, passwo
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
                    WebTarget target = client.target(restURL);
                    target.configuration().register(new HttpBasicAuthFilter(userna
```

18.2.3. Setting "Accept" header

Jersey 1.x way:

```
Client client = Client.create();
                    WebResource webResource = client.resource(restURL).accept("tex
                    ClientResponse response = webResource.get(ClientResponse.class
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
                    WebTarget target = client.target(restURL);
                    Response response = target.request("text/plain").get(Response.
```

18.2.4. Attaching entity to request

Jersey 1.x way:

```
Client client = Client.create();
                    WebResource webResource = client.resource(restURL);
                    ClientResponse response = webResource.post(ClientResponse.clas
```

JAX-RS 2.0 way:

```
Client client = ClientFactory.newClient();
                    WebTarget target = client.target(restURL);
                    Response response = target.request().post(Entity.text("payload
```

18.2.5. Setting SSLContext and/or HostnameVerifier

Jersey 1.x way:

```
HTTPSProperties prop = new HTTPSProperties(hostnameVerifier, sslContext);
                    DefaultClientConfig dcc = new DefaultClientConfig();
                    dcc.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERT
                    Client client = Client.create(dcc);
```

Jersey 2.0 way:

```
Client client = ClientFactory.newClient();
                    client.configuration().setProperty(ClientProperties.SSL_CONTEX
                    client.configuration().setProperty(ClientProperties.HOSTNAME_V
```