Microsoft

Mobile Application Architecture Guide

Application Architecture Pocket Guide Series

patterns & practices

Mobile Application Architecture Guide

patterns & practices

J.D. Meier
Alex Homer
David Hill
Jason Taylor
Prashant Bansode
Lonnie Wall
Rob Boucher Jr
Akshay Bogawat

Introduction

Overview

The purpose of the Mobile Application Architecture Pocket Guide is to improve your effectiveness when building mobile applications on the Microsoft platform. The primary audience is solution architects and development leads. The guide provides design-level guidance for the architecture and design of mobile applications built on the .NET Platform. It focuses on partitioning application functionality into layers, components, and services, and walks through their key design characteristics.

The guidance is task-based and presented in chapters that correspond to major architecture and design focus points. It is designed to be used as a reference resource, or it can be read from beginning to end. The guide contains the following chapters and resources:

- **Chapter 1, "Mobile Application Architecture,"** provides general design guidelines for a mobile application, explains the key attributes, discusses the use of layers, provides guidelines for performance, security, and deployment, and lists the key patterns and technology considerations.
- **Chapter 2, "Architecture and Design Guidelines,"** helps you to understand the concepts of software architecture, learn the key design principles for software architecture, and provides the guidelines for the key attributes of software architecture.
- **Chapter 3, "Presentation Layer Guidelines,"** helps you to understand how the presentation layer fits into the typical application architecture, learn about the components of the presentation layer, learn how to design these components, and understand the common issues faced when designing a presentation layer. It also contains key guidelines for designing a presentation layer, and lists the key patterns and technology considerations.
- **Chapter 4, "Business Layers Guidelines,"** helps you to understand how the business layer fits into the typical application architecture, learn about the components of the business layer, learn how to design these components, and understand common issues faced when designing a business layer. It also contains key guidelines for designing the business layer, and lists the key patterns and technology considerations.
- **Chapter 5, "Data Access Layer Guidelines,"** helps you top understand how the data layer fits into the typical application architecture, learn about the components of the data layer, learn how to design these components, and understand the common issues faced when designing a data layer. It also contains key guidelines for designing a data layer, and lists the key patterns and technology considerations.
- **Chapter 6, "Service Layer Guidelines,"** helps you to understand how the service layer fits into the typical application architecture, learn about the components of the service layer, learn how to design these components, and understand common issues faced when designing a service layer. It also contains key guidelines for designing a service layer, and lists the key patterns and technology considerations.
- **Chapter 7, "Communication Guidelines,"** helps you to learn the guidelines for designing a communication approach, and understand the ways in which components communicate

with each other. It will also help you to learn the interoperability, performance, and security considerations for choosing a communication approach, and the communication technology choices available.
- **Chapter 8, "Deployment Patterns,"** helps you to learn the key factors that influence deployment choices, and contains recommendations for choosing a deployment pattern. It also helps you to understand the effect of deployment strategy on performance, security, and other quality attributes, and learn common deployment patterns.

Why We Wrote This Guide

We wrote this guide to accomplish the following:
- To help you design more effective architectures on the .NET platform.
- To help you choose the right technologies
- To help you make more effective choices for key engineering decisions.
- To help you map appropriate strategies and patterns.
- To help you map relevant patterns & practices solution assets.

Features of This Guide

- **Framework for application architecture**. The guide provides a framework that helps you to think about your application architecture approaches and practices.
- **Architecture Frame**. The guide uses a frame to organize the key architecture and design decision points into categories, where your choices have a major impact on the success of your application.
- **Principles and practices**. These serve as the foundation for the guide, and provide a stable basis for recommendations. They also reflect successful approaches used in the field.
- **Modular**. Each chapter within the guide is designed to be read independently. You do not need to read the guide from beginning to end to get the benefits. Feel free to use just the parts you need.
- **Holistic**. If you do read the guide from beginning to end, it is organized to fit together. The guide, in its entirety, is better than the sum of its parts.
- **Subject matter expertise**. The guide exposes insight from various experts throughout Microsoft, and from customers in the field.
- **Validation**. The guidance is validated internally through testing. In addition, product, field, and support teams have performed extensive reviews. Externally, the guidance is validated through community participation and extensive customer feedback cycles.
- **What to do, why, how**. Each section in the guide presents a set of recommendations. At the start of each section, the guidelines are summarized using bold, bulleted lists. This gives you a snapshot view of the recommendations. Then each recommendation is expanded to help you understand what to do, why, and how.
- **Technology matrices**. The guide contains a number of cheat sheets that explore key topics in more depth. Use these cheat sheets to help you make better decisions on technologies, architecture styles, communication strategies, deployment strategies, and common design patterns.

- **Checklists**. The guide contains checklists for communication strategy as well as each mobile application layer. Use these checklists to review your design as input to drive architecture and design reviews for your application.

Audience

This guide is useful to anyone who cares about application design and architecture. The primary audience for this guide is solution architects and development leads, but any technologist who wants to understand good application design on the .NET platform will benefit from reading it.

Ways to Use the Guide

You can use this comprehensive guidance in several ways, both as you learn more about the architectural process and as a way to instill knowledge in the members of your team. The following are some ideas:

- **Use it as a reference**. Use the guide as a reference and learn the architecture and design practices for mobile applications on the .NET Framework.
- **Use it as a mentor**. Use the guide as your mentor for learning how to design an application that meets your business goals and quality attributes objectives. The guide encapsulates the lessons learned and experience from many subject-matter experts.
- **Use it when you design applications**. Design applications using the principles and practices in the guide, and benefit from the lessons learned.
- **Create training**. Create training from the concepts and techniques used throughout the guide, as well as from the technical insight into the .NET Framework technologies.

Feedback and Support

We have made every effort to ensure the accuracy of this guide. However, we welcome feedback on any topics it contains. This includes technical issues specific to the recommendations, usefulness and usability issues, and writing and editing issues.

If you have comments on this guide, please visit the Application Architecture KB at http://www.codeplex.com/AppArch.

Technical Support

Technical support for the Microsoft products and technologies referenced in this guidance is provided by Microsoft Product Support Services (PSS). For product support information, please visit the Microsoft Product Support Web site at: http://support.microsoft.com.

Community and Newsgroup Support

You can also obtain community support, discuss this guide, and provide feedback by visiting the MSDN Newsgroups site at http://msdn.microsoft.com/newsgroups/default.asp.

The Team Who Brought You This Guide

This guide was produced by the following .NET architecture and development specialists:

- J.D. Meier
- Alex Homer
- David Hill
- Jason Taylor
- Prashant Bansode
- Lonnie Wall
- Rob Boucher Jr.
- Akshay Bogawat

Contributors and Reviewers

- **Test Team**. Rohit Sharma; Praveen Rangarajan
- **Edit Team**. Dennis Rea
- **Special Thanks**: Rob Tiffany; Rabi Satter

Tell Us About Your Success

If this guide helps you, we would like to know. Tell us by writing a short summary of the problems you faced and how this guide helped you out. Submit your summary to MyStory@Microsoft.com.

Chapter 1: Mobile Application Architecture

Objectives

- Define a mobile application.
- Understand components found in a mobile application.
- Learn the key scenarios where mobile applications would be used.
- Learn the design considerations for mobile applications.
- Identify specific scenarios for mobile applications, such as deployment, power usage, and synchronization.
- Learn the key patterns and technology considerations for designing mobile applications.

Overview

A mobile application will normally be structured as a multi-layered application consisting of user experience, business, and data layers. When developing a mobile application, you may choose to develop a thin Web-based client or a rich client. If you are building a rich client, the business and data services layers are likely to be located on the device itself. If you are building a thin client, the business and data layers will be located on the server. Figure 1 illustrates common rich client mobile application architecture with components grouped by areas of concern.

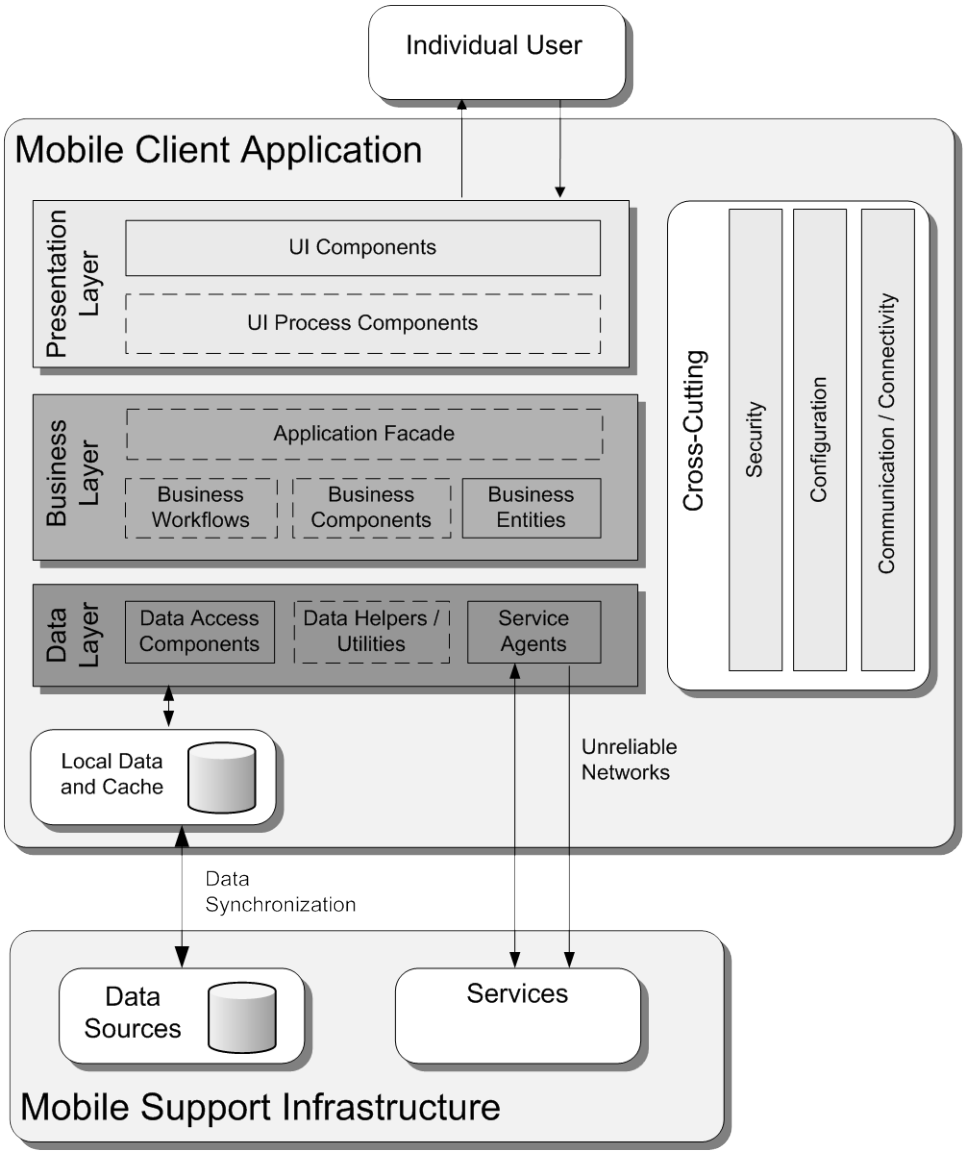

Figure 1 Common rich client mobile application architecture

Design Considerations

The following design guidelines provide information about different aspects you should consider when designing a mobile application. Follow these guidelines to ensure that your application meets your requirements and performs efficiently in scenarios common to mobile applications:

- **Decide if you will build a rich client, a thin Web client, or rich Internet application (RIA)**. If your application requires local processing and must work in an occasionally connected scenario, consider designing a rich client. A rich client application will will be more complex to install and maintain. If your application can depend on server processing and will always be fully connected, consider designing a thin client. If your application requires a rich user

interface (UI), only limited access to local resources, and must be portable to other platforms, design an RIA client.

- **Determine the device types you will support**. When choosing which device types to support, consider screen size, resolution (DPI), CPU performance characteristics, memory and storage space, and development tool environment availability. In addition, factor in user requirements and organizational constraints. You may require specific hardware such as GPS or a camera and this may impact not only your application type, but also your device choice.
- **Design considering occasionally connected limited-bandwidth scenarios when required.** If your mobile device is a stand-alone device, you will not need to account for connection issues. When network connectivity is required, Mobile applications should handle cases when a network connection is intermittent or not available. It is vital in this case to design your caching, state management, and data-access mechanisms with intermittent network connectivity in mind. Batch communications for times of connectivity. Choose hardware and software protocols based on speed, power consumption, and "chattiness," and not just on ease of programming
- **Design a UI appropriate for mobile devices, taking into account platform constraints**. Mobile devices require a simpler architecture, simpler UI, and other specific design decisions in order to work within the constraints imposed by the device hardware. Keep these constraints in mind and design specifically for the device instead of trying to reuse the architecture or UI from a desktop or Web application. The main constraints are memory, battery life, ability to adapt to difference screen sizes and orientations, security, and network bandwidth.
- **Design a layered architecture appropriate for mobile devices that improves reuse and maintainability**. Depending on the application type, multiple layers may be located on the device itself. Use the concept of layers to maximize separation of concerns, and to improve reuse and maintainability for your mobile application. However, aim to achieve the smallest footprint on the device by simplifying your design compared to a desktop or Web application.
- **Design considering device resource constraints such as battery life, memory size, and processor speed**. Every design decision should take into account the limited CPU, memory, storage capacity, and battery life of mobile devices. Battery life is usually the most limiting factor in mobile devices. Backlighting, reading and writing to memory, wireless connections, specialized hardware, and processor speed all have an impact on the overall power usage. When the amount of memory available is low, the Microsoft® Windows Mobile® operating system may ask your application to shut down or sacrifice cached data, slowing program execution. Optimize your application to minimize its power and memory footprint while considering performance during this process.

Mobile Client Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Table 1 Mobile Client Frame

Category	Key issues
Authentication and Authorization	• Failing to authenticate in occasionally connected scenarios • Failing to authorize in occasionally-connected scenarios • Failing to use authentication and authorization over a virtual private network (VPN) • Failing to authenticate during synchronization over the air • Failing to authenticate during synchronization with the host PC • Failing to authenticate for all connection scenarios, such as over the air, cradled**,** Bluetooth, and Secure Digital (SD) cards • Failing to appreciate the differences between security models of different devices
Caching	• Caching unnecessary data on a device that has limited resources • Relying on cached data that may no longer be available in occasionally-connected scenarios • Choosing inappropriate cache locations and formats • Caching sensitive data in unencrypted form • Failing to choose an appropriate caching technology
Communication	• Failing to protect sensitive data over the air • Failing to secure Web service communication • Failing to secure communication over a VPN • Not appreciating the performance impact of communication security on limited-bandwidth connections • Not managing limited-bandwidth connections efficiently • Not managing connections to multiple network services efficiently • Not designing to work with intermittent connectivity • Not considering connection cost or allowing the user to manage connections • Not designing to minimize power usage when running on battery power • Failing to use the appropriate communication protocol
Configuration Management	• Failing to restore configuration state after a reset • Failing to consider configuration management synchronization over the air • Failing to consider configuration management synchronization with the host PC • Choosing an inappropriate format for configuration information • Failing to protect sensitive configuration information • Failing to consider the techniques used by different manufacturers for loading configuration settings
Data Access	• Failing to implement data-access mechanisms that work with intermittent connectivity • Not considering database access performance • Navigating through large datasets when not required

Category	Key issues
	• Failing to consider appropriate replication technologies and techniques • Failing to consider access to device database services such as Microsoft SQL Server® Compact Edition
Device	• Failing to consider device heterogeneity, such as screen size and CPU power • Not presenting user-friendly error messages to the user • Failing to protect sensitive information • Failure to consider the processing power of the device
Exception Management	• Not recovering application state after an exception • Revealing sensitive information to the end user • Not logging sufficient details about the exception • Using exceptions to control application flow
Logging	• Not considering remote logging instead of logging on the device • Not considering how to access device logs • Not considering resource constraints when logging • Failing to protect sensitive information in the log files
Porting	• Failing to rewrite the existing rich client UI to suit the device • Failing to explore the available porting tools
Synchronization	• Failing to secure synchronization when communicating • Failing to manage synchronization over the air as opposed to cradled synchronization • Failing to manage synchronization interruptions • Failing to handle synchronization conflicts • Failing to consider merge replication where appropriate
Testing	• Failing to appreciate debugging costs when choosing to support multiple device types • Failing to design with debugging in mind; for example, using emulators instead of the actual devices • Failing to debug in all connection scenarios
UI	• Not considering the restricted UI form factor • Not considering the single window environment • Not considering that only one application can be running • Not designing a touch-screen or stylus-driven UI for usability • Not including support for multiple screen sizes and orientations • Not managing device reset and resume • Not considering the limited API and reduced range of UI controls compared to the desktop
Validation	• Not validating input and data during host PC communication • Not validating input and data during over-the-air communication • Failing to protect hardware resources, such as the camera and initiation of phone calls • Not designing validation with limited resources and performance in mind

Authentication and Authorization

Designing an effective authentication and authorization strategy is important for the security and reliability of your application. Weak authentication can leave your application vulnerable to unauthorized use. Mobile devices are usually designed to be single-user devices and normally lack basic user profile and security tracking beyond just a simple password. Other common desktop mechanisms are also likely to be missing. The discoverability of mobile devices over protocols such as Bluetooth can present users with unexpected scenarios. Mobile applications can also be especially challenging due to connectivity interruptions. Consider all possible connectivity scenarios, whether over-the-air or hard-wired.

Consider the following guidelines when designing authentication and authorization:

- Design authentication for over-the-air, cradled synchronization, Bluetooth discovery, and local SD card scenarios.
- Consider that different devices might have variations in their programming security models, which can affect authorization to access resources
- Do not assume that security mechanisms available on larger platforms will be available on a mobile platform, even if you are using the same tools. For example, access control lists (ACLs) are not available in Windows Mobile, and consequently there is no operating system–level file security.
- Ensure that you require authentication for access by Bluetooth devices.
- Identify trust boundaries within your mobile application layers; for instance, between the client and the server or the server and the database. This will help you to determine where and how to authenticate.

Caching

Use caching to improve the performance and responsiveness of your application, and to support operation when there is no network connection. Use caching to optimize reference data lookups, to avoid network round trips, and to avoid unnecessarily duplicated processing. When deciding what data to cache, consider the limited resources of the device; you will have less storage space available than on a PC.

Consider the following guidelines when designing caching:

- Identify your performance objectives. For example, determine your minimum response time and battery life. Test the performance of the specific devices you will be using. Most mobile devices use only flash memory, which is likely to be slower than the memory used in desktop machines.
- Design for minimum memory footprint. Cache only data that is absolutely necessary for the application to function, or expensive to transform into a ready-to-use format. If designing a memory-intensive application, detect low-memory scenarios and design a mechanism for prioritizing the data to discard as available memory decreases.
- Cache static data that is useful, and avoid caching volatile data. However, consider caching any data, including volatile data, that the application will need in an occasionally connected or offline scenario.

- Consider using SQL Server Compact edition for caching instead of device memory. Memory consumed by the application may be cleared in low memory situations.
- Choose the appropriate cache location, such as on the device, at the mobile gateway, or in the database server.

Communication

Device communication includes wireless communication (over the air) and wired communication with a host PC, as well as more specialized communication such as Bluetooth or Infrared Data Association (IrDA). When communicating over the air, consider data security to protect sensitive data from theft or tampering. If you are communicating through Web service interfaces, use mechanisms such as the WS-Secure standards to secure the data. Keep in mind that wireless device communication is more likely to be interrupted than communication from a PC, and that your application might be required to operate for long periods in a disconnected state.

Consider the following guidelines when designing your communication strategy:

- Design asynchronous, threaded communication to improve usability in occasionally connected scenarios.
- If you are designing an application that will run on a mobile phone, consider the effects of receiving a phone call during communication or program execution. Design the application to allow it to suspend and resume, or even exit the application.
- Protect communication over untrusted connections, such as Web services and other over-the-air methods.
- If you must access data from multiple sources, interoperate with other applications, or work while disconnected, consider using Web services for communication.
- If you are using WCF and for communication and need to implement message queuing, consider using WCF store and forward.

Configuration Management

When designing device configuration management, consider how to handle device resets, as well as whether you want to allow configuration of your application over the air or from a host PC.

Consider the following guidelines when designing your configuration-management strategy:

- Design for the restoration of configuration after a device reset.
- If you have you enterprise data in Microsoft SQL Server 2005 or 2008 and desire an accelerated time to market, consider using merge replication with a “buy and configure” application from a third party. Merge replication can synchronize data in a single operation regardless of network bandwidth or data size.
- Due to memory limitations, choose binary format over Extensible Markup Language (XML) for configuration files
- Protect sensitive data in device configuration files.

- Consider using compression library routines to reduce the memory requirements for configuration and state information.
- If you have a Microsoft Active Directory® directory service infrastructure, consider using the System Center Mobile Device Manager interface to manage group configuration, authentication, and authorization of devices. See the Technology Considerations section for requirements for the Mobile Device Manager.

Data Access

Data access on a mobile device is constrained by unreliable network connections and the hardware constraints of the device itself. When designing data access, consider how low bandwidth, high latency, and intermittent connectivity will impact your design.

Consider the following guidelines when designing data access:

- Always prefer strongly-typed collections or generics over DataSets and XML due to to reduce memory overhead and improve performance. If you decide to use DataSets and are only reading (and not writing) data, utilize **DataReader**s.
- Consider using a local device database that provides synchronization services, such as SQL Server Compact Edition. Only architect a special mechanism to synchronize data if the standard data synchronization cannot meet your requirements.
- Program for data integrity. Files left open during device suspend and power failures may cause data-integrity issues, especially when data is stored on a removable storage device. Include exception handling and retry logic to ensure that file operations succeed.
- Do not assume that removable storage will always be available, as a user can remove it at any time. Check for the existence of a removable storage device before writing or using FlushFileBuffers.
- If you need to ensure data integrity in case the device loses power or has connectivity disruptions, considering using transactions with SQL Server Mobile.
- If you use XML to store or transfer data, consider its overall size and impact on performance. XML increases both bandwidth and local storage requirements. Use compression algorithms or a non-XML transfer method.
- If your application needs to sync with multiple database types, use Sync Services for ADO.NET. It allows you to store data in Microsoft SQL Server, Oracle, or DB2.

Device

Mobile device design and development is unique due to the constrained and differing nature of device hardware. You may be targeting multiple devices with very different hardware parameters. Keep the heterogeneous device environment in mind when designing your mobile application. Factors include variations in screen size and orientation, limitations in memory and storage space, and network bandwidth and connectivity. Your choice of a mobile operating system will generally depend on the target device type.

Consider the following guidelines when determining your device strategy:

- Optimize the application for the device by considering factors such as screen size and orientation, network bandwidth, memory storage space, processor performance, and other hardware capabilities.
- Consider device-specific capabilities that you can use to enhance your application functionality, such as accelerometers, graphics processing units (GPUs), global positioning systems (GPS), haptic (touch, force and vibration) feedback, compass, camera, and fingerprint readers.
- If you are developing for more than one device, design first for the subset of functionality that exists on all of the devices, and then customize for device-specific features when they are detected.
- Create modular code to allow easy module removal from executables. This covers cases where separate smaller executable files are required due to device memory-size constraints.

Exception Management

Designing an effective exception-management strategy is important for the security and reliability of your application. Good exception handling in your mobile application prevents sensitive exception details from being revealed to the user, improves application robustness, and helps to avoid your application being left in an inconsistent state in the event of an error.

Consider the following guidelines when designing for exception management:

- Design your application to recover to a known good state after an exception occurs.
- Do not use exceptions to control logic flow.
- Do not catch exceptions unless you can handle them.
- Design a global error handler to catch unhandled exceptions.
- Design an appropriate logging and notification strategy that does not reveal sensitive information for critical errors and exceptions.

Logging

Because of the limited memory on mobile devices, logging and instrumentation should be limited to only the most necessary cases; for example, attempted intrusion into the device. When devices are designed to be a part of a larger infrastructure, choose to track most device activity at the infrastructure level. Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access, and by the same routines that access the resource. Consider the fact that some of the logs might have to be generated on the device and must be synchronized with the server during periods of network connectivity.

Consider the following guidelines when designing logging:

- If you carry out extensive logging on the device, consider logging in an abbreviated or compressed format to minimize memory and storage impact. There is no system Event Log in Windows Mobile.
- Consider using an 3rd party logging mechanism that supports the .NET Compact Framework. Several exist at the time this document was written. (OpenNetCF, nLog, log4Net)

- Consider using platform features such as health monitoring on the server, and mobile device services on the device, to log and audit events. Explore adding in remote health monitoring capabilities using the Open Mobile Alliance Device Management (OMA DM) standard.
- Synchronize between the mobile database logs and the server database logs to maintain audit capabilities on the server. If you have an Active Directory infrastructure, consider using the System Center Mobile Device Manager to extract logs from mobile devices. See the Technology Considerations section for requirements for the Mobile Device Manager.
- Do not store sensitive information in log files.
- Decide what constitutes unusual or suspicious activity on a device, and log information based on these scenarios.

Porting

Developers often want to port part or all of an existing application to a mobile device. Certain types of applications will be easier to port than others, and it is unlikely that you will be able to port the code directly without modification.

Consider the following guidelines when designing to port your existing application to a mobile device:

- If you are porting a rich client application from the desktop, rewrite the application in its entirety. Rich clients are rarely designed to suit a small screen size and limited memory and disk resources.
- If you are porting a Web application to a mobile device, consider rewriting the UI for the smaller screen size. Also, consider communication limitations and interface chattiness as these can translate into increased power usage and connection costs for the user.
- If you are porting an RIA client, research details to discover which code will port without modification. Consult the technology considerations section of this chapter for specific advice.
- Research and utilize tools to assist in porting. For example, Java-to-C++ convertors are available. When converting from Smartphone to Pocket PC code, Microsoft Visual Studio® will allow you to change the target platform and will provide warnings when you are using Smartphone-specific functionality. You can also link Visual Studio Desktop and Mobile projects to assist in knowing what is portable between the two projects.
- Do not assume that you can port custom controls as-is to a mobile application. Supported APIs, memory footprint, and UI behavior are different on a mobile device. Test the controls as early as possible so that you can plan to rewrite them or find an alternative if required.

Power

Power is the most limiting factor for a mobile device. All design decisions should at take into account how much power the device consumes and its effect on overall battery life. If you have a choice in devices, consider devices that can draw power from Universal Serial Bus (USB) or other types of data hookups. Research communication protocols for their power consumption.

Consider the following guidelines when designing for power consumption:

- To conserve battery life, do not update the UI while the application is in the background.
- Choose communication methods considering both power usage as well as network speed.
- Consider deferring nonessential wireless communications until the device is under external power.
- Implement power profiles to increase performance when the device is plugged into external power and not charging its battery.
- Design so that parts of the devices can be powered down when not in use, or when not required. Common examples are screen backlighting, hard drives, GPS functions, speakers, and wireless communications.
- Design services and communications to transfer the smallest number of bytes possible over the air. Choose protocols, design service interfaces, and batch communications with this goal in mind.
- If you are considering using the 3G hardware communications protocol, consider that while it is significantly faster, it also currently uses much more power than its predecessors, such as the Edge protocol. When you are using 3G, be sure to communicate in batched bursts and to shut down communication at times when it is not needed.

Synchronization

Consider whether you want to support over-the-air synchronization, cradled synchronization, or both. Because synchronization will often involve sensitive data, consider how to secure your synchronization data, especially when synchronizing over the air. Design your synchronization to handle connection interruptions gracefully, either by canceling the operation or by allowing it to resume when a connection becomes available. Merge replication allows both upload-only and bidirectional synchronization and is a good choice for infrastructures utilizing newer versions of SQL Server. Consider the Microsoft Sync Framework as it can provide robust synchronization services in a wide variety of situations.

Consider the following guidelines when designing synchronization:

- Design for recovery when synchronization is reset, and decide how to manage synchronization conflicts.
- If you must support bidirectional synchronization to SQL Server, consider using merge replication synchronization. Remember that merge synchronization will synchronize all of the data in the merge set, which may require additional network bandwidth and can impact performance.
- If your users must synchronize data when away from the office, consider including over-the-air synchronization in your design.
- If your users will be synchronizing with a host PC, consider including cradled synchronization in your design.
- Consider store-and-forward synchronization using WCF rather than e-mail or SMS (text message), as WCF guarantees delivery and works well in a partially connected scenario.

Testing

Mobile application debugging can be much more expensive than debugging a similar application on a PC. Consider this debugging cost when deciding which devices, and how many devices, your application will support. Also keep in mind that it can be harder to get debug information from the device, and that device emulators do not always perfectly simulate the device hardware environment.

Consider the following guidelines when designing your debugging strategy:

- Understand your debugging costs when choosing which devices to support. Factor in tools support, the cost of initial (and perhaps replacement) test devices, and the cost of software-based device emulators.
- If you have the device you are targeting, debug your code on the actual device rather than on an emulator.
- If the device is not available, use an emulator for initial testing and debugging. Consider that an emulator might run code slower than the actual device. As soon as you obtain the device, switch to running code on the device connected to a normal PC. Perform final testing on your device when not connected to a PC. Add in temporary or permanent mechanisms to debug problems in this scenario. Consider the needs of people who will support the device.
- Test scenarios where your device is fully disconnected from any network or connection, including being disconnected from a PC debugging session.
- If you are an OEM or ODM and your device has not yet been created, note that it is possible to debug a mobile program on a dedicated x86-based Microsoft Windows® CE PC. Consider this option until your device is available.

User Interface

When designing the UI for a mobile application, do not try to adapt or reuse the UI from a desktop application. Design your device UI so that it is as simple as possible, and designed specifically for pen-based input and limited data entry capabilities as appropriate. Consider the fact that your mobile application will run in full-screen mode and will only be able to display a single window at a time. Therefore, blocking operations will prevent the user from interacting with the application. Consider the various screen sizes and orientations of your target devices when designing your application UI.

Consider the following guidelines when designing the UI for your mobile application:

- Design considering that a person’s hand can block a touch-screen UI during input with a stylus or finger. For example, place menu bars at the bottom of the screen, expanding options upwards.
- Design for a single-window, full-screen UI. If your device will be a single-user device running only the main application, consider using kiosk mode. Keep in mind that Windows Mobile does not support a kiosk mode, so you will need to use Windows CE.
- Consider input from various sources such as stylus, keypad, and touch. For example, accommodate touch-screen input by making buttons large enough, and lay out controls so

that the UI is usable using a finger or stylus for input. Design for various screen sizes and orientations.
- Give the user visual indication of blocking operations; for example, an hourglass cursor.

Performance Considerations

Design your mobile application with device hardware and performance constraints in mind. Designing for a mobile device requires that you consider limited CPU speed, reduced memory and storage, narrow bandwidth and high latency connections, and limited battery life.

Consider the following guidelines when designing your performance strategy:
- Design configurable options to allow the maximum use of device capabilities. Allow users to turn off features they do not require in order to save power.
- To optimize for mobile device resource constraints, consider using lazy initialization.
- Consider limited memory resources and optimize your application to use the minimum amount of memory. When memory is low, the system may release cached intermediate language (IL) code to reduce its own memory footprint, return to interpreted mode, and thus slow overall execution.
- Consider using programming shortcuts as opposed to following pure programming practices that can inflate code size and memory consumption. For example, examine the cost of using pure object-oriented practices such as abstract base classes and repeated object encapsulation.
- Consider power consumption when using the device CPU, wireless communication, screen, or other power-consuming resources while on battery power. Balance performance with power consumption.

Deployment

Mobile applications can be deployed using many different methods. Consider the requirements of your users, as well as how you will manage the application, when designing for deployment. Ensure that you design to allow for the appropriate management, administration, and security for application deployment.

Deployment scenarios listed for Windows Mobile device applications, with the more common ones listed first, are:
- Microsoft Exchange ActiveSync® using a Windows Installer file (MSI).
- Over the air, using HTTP, SMS, or CAB files to provide install and run functionality.
- Mobile Device Manager–based, using Active Directory to load from a CAB or MSI file.
- Post load and auto-run, which loads a company-specific package as part of the operating system.
- Site loading, manually using an SD card.

Consider the following guidelines when designing your deployment strategy:
- If your users must be able to install and update applications while away from the office, consider designing for over-the-air deployment.

- If you are using CAB file distribution for multiple devices, include multiple device executables in the CAB file. Have the device detect which executable to install, and discard the rest.
- If your application relies heavily on a host PC, consider using ActiveSync to deploy your application.
- If you are deploying a baseline experience running on top of Windows Mobile, considering using the post-load mechanism to automatically load your application immediately after the Windows Mobile operating system starts up.
- If your application will be run only at a specific site, and you want to manually control distribution, consider deployment using an SD memory card.

Pattern Map

Key patterns are organized by the key categories detailed in the Mobile Client Frame in the following table. Consider using these patterns when making design decisions for each category.

Table 2 Pattern Map

Category	Relevant patterns
Caching	• Lazy Acquisition
Communication	• Active Object • Communicator • Entity Translator • Reliable Sessions
Data Access	• Active Record • Data Transfer Object • Domain Model • Transaction Script
Synchronization	• Synchronization
UI	• Application Controller • Model-View-Controller • Model-View-Presenter • Pagination

- For more information on *Application Controller*, *Model View Controller (MVC), Domain Model*, *Transaction Script*, *Active Record*, *Data Mapper*, *Data Transfer Object*, patterns see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/
- For more information on *Entity Translator* pattern see "Useful Patterns for Services" at http://msdn.microsoft.com/en-us/library/cc304800.aspx
- For more information on Active Object pattern see "Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects." Published by Chichester, England; New York: John Wiley & Sons Ltd., 2000

- For more information on Communicator pattern see “Patterns for point-to-point communications” by Guidi-Polanco, F., Cubillos F., C., Menga, G., & Penha, S. (2003)
- For more information on Pagination pattern see “Improving Application Performance by Implementing Paginated Lists” at http://msdn.microsoft.com/en-us/library/ms978330.aspx
- For more information on synchronization pattern see “Data Patterns” at http://msdn.microsoft.com/en-us/library/ms998446.aspx
- For more information on Lazy Acquisition pattern see “Pattern-Oriented Software Architecture, Patterns for Resource Management. John Wiley & Sons.”

Pattern Descriptions

- **Active Object**. Support asynchronous processing by encapsulating the service request and service completion response.
- **Active Record**. Include a data access object within a domain entity.
- **Application Controller**. An object that contains all of the flow logic, and is used by other Controllers that work with a Model and display the appropriate View.
- **Communicator**. Encapsulate the internal details of communication in a separate component that can communicate through different channels.
- **Data Transfer Object (DTO)**. An object that stores the data transported between processes, reducing the number of method calls required.
- **Domain Model**. A set of business objects that represents the entities in a domain and the relationships between them.
- **Entity Translator**. An object that transforms message data types into business types for requests, and reverses the transformation for responses.
- **Lazy Acquisition**. Defer the acquisition of resources as long as possible to optimize device resource use.
- **Model-View-Controller**. Separate the UI code into three separate units: Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Model-View-Presenter**. Separate request processing into three separate roles, with the View being responsible for handling user input and passing control to a Presenter object.
- **Pagination**. Separate large amounts of content into individual pages to optimize system resources and minimize use of screen space.
- **Reliable Sessions**. End-to-end reliable transfer of messages between a source and a destination, regardless of the number or type of intermediaries that separate the endpoints
- **Synchronization**. A component installed on a device tracks changes to data and exchanges information with a component on the server when a connection is available.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

The following guidelines contain suggestions and advice for common scenarios for mobile applications and technologies.

Microsoft Silverlight for Mobile

Consider the following guidelines if you are using Microsoft Silverlight® for Mobile:

- At the time of document's release, Silverlight for Mobile was an announced product under development, but not yet released.
- If you want to build applications that support rich media and interactivity and have the ability to run on both a mobile device and desktop as is, consider using Silverlight for Mobile. Silverlight 2.0 code created to run on the desktop in the Silverlight 2.0 plug-in will run in the Windows Mobile Silverlight plug-in in the latest version of Microsoft Internet Explorer for Mobile. Consider that while it is possible to use the same Silverlight code in both places, you should take into account the differing screen size and resource constraints on a mobile device. Consider optimizing the code for Windows Mobile.
- If you want to develop Web pages for both desktop and mobile platforms, consider Silverlight for Mobile or normal ASP.NET/HMTL over ASP.NET for Mobile unless you know that your device cannot support either of these alternatives. As device browsers have become more powerful, they are able to process the same native HTML and ASP.NET targeted by the desktop, thus making ASP.NET Mobile development less important. ASP.NET Mobile Controls currently supports a variety of mobile devices through specific markup adapters and device profiles. While ASP.NET Mobile Controls automatically render content to match device capabilities at run time, there is overhead associated with testing and maintaining the device profiles. Development support for these controls is included in Microsoft Visual Studio 2003 and 2005 but is no longer supported in Visual Studio 2008. Run-time support is currently still available, but may be discontinued in the future. For more information, see the links available in the Additional Resources section.

.NET Compact Framework

Consider the following guidelines if you are using the Microsoft .NET Compact Framework:

- If you are familiar with the Microsoft .NET Framework and are developing for both the desktop and mobile platforms concurrently, consider that the .NET Compact Framework is a subset of the .NET Framework class library. It also contains some classes exclusively designed for Windows Mobile. The .NET Compact Framework supports only Microsoft Visual Basic® and Microsoft Visual C#® development.
- If you are attempting to port code that uses Microsoft Foundation Classes (MFC), consider that it is not trivial due to MFC's dependency on Object Linking and Embedding (OLE). The Windows Compact Edition supports COM, but not OLE. Check to see if the OLE libraries are available for separate download to your device before trying to use MFC on a mobile device.

- If you have issues tracing into a subset of Windows Mobile code with the Visual Studio debugger, consider that you might require multiple debug sessions. For example, if you have both native and managed code in the same debug session, Visual Studio might not follow the session across the boundary. In this case, you will require two instances of Visual Studio running and will have to track the context between them manually.

Windows Mobile

Consider the following general guidelines for Windows Mobile applications:

- If you are targeting an application for both Windows Mobile Professional and Windows Mobile Standard editions, consider that the security model varies on the different versions of Windows Mobile. Code that works on one platform might not work on the other because of the differing security models for APIs. Check the Windows Mobile documentation for your device and version. See the Additional Resources section.
- If you will have to manage your application in the future or are upgrading an existing application, be sure that you understand the Windows Mobile operating system derivation, product naming, and versioning tree. There are slight differences between each version that could potentially impact your application.
    - Windows Mobile is derived from releases of the Windows CE operating system.
    - Both Windows Mobile version 5.x and 6.x are based on Windows CE version 5.x.
    - Windows Mobile Pocket PC was renamed Windows Mobile Professional starting with Windows Mobile 6.0
    - Windows Mobile Smartphone was renamed Windows Mobile Standard starting with Windows Mobile 6.0.
    - Windows Mobile Professional and Windows Mobile Standard have slight differences in their APIs. For example, the Windows Mobile Standard (Smartphone) lacks a Button class in its Compact Framework implementation because softkeys are used for data entry instead.
- Always use the Windows Mobile APIs to access memory and file structures. Do not access them directly after you have obtained a handle to either structure. Windows CE version 6.x (and thus the next release of Windows Mobile) uses a virtualized memory model and a different process execution model than previous versions. This means that structures such as file handles and pointers may no longer be actual physical pointers to memory. Windows Mobile programs that relied on this implementation detail in versions 6.x and before will fail when moved to the next version of Windows Mobile.
- The Mobile Device Manager is mentioned in this article as a possible solution for authorizing, tracking, and collecting logs from mobile devices, assuming that you have an Active Directory infrastructure. MDM also requires a number of other products to fully function, including:
    - Windows Mobile 6.1 on devices
    - Windows Software Update Service (WSUS) 3.0
    - Windows Mobile Device Management Server
    - Enrollment Server

- Gateway Server
- Active Directory as part of Windows Server
- SQL Server 2005 or above
- Microsoft Certificate Authority
- Internet Information Server (IIS) 6.0
- .NET Framework 2.0 or above

Windows Embedded

Consider the following guidelines if you are choosing a Windows Embedded technology:

- If you are designing for a set-top box or other larger-footprint device, consider using Windows Embedded Standard.
- If you are designing for a point-of-service (POS) device such as an automated teller machine (ATMs, customer-facing kiosks, or self-checkout systems), consider using Windows Embedded for Point of Service.
- If you are designing for a GPS-enabled device or a device with navigation capabilities, consider using Microsoft Windows Embedded NavReady™. Note that Windows Embedded NavReady 2009 is built on Windows Mobile 5.0, while Windows Mobile version 6.1 is used in the latest versions for Windows Mobile Standard and Professional. If you are targeting a common codebase for NavReady and other Windows Mobile devices, be sure to verify that you are using APIs available on both platforms.

Additional Resources

- For more information on the Windows Embedded technology options, see the Windows Embedded Developer Center at http://msdn.microsoft.com/en-us/embedded/default.aspx.
- For the patterns & practices Mobile Client Software Factory, see http://msdn.microsoft.com/en-us/library/aa480471.aspx
- For information on the Microsoft Sync Framework, see http://msdn.microsoft.com/en-us/sync/default.aspx
- For more information on the OpenNETCF.Diagnostics.EventLog in the Smart Device Framework see http://msdn.microsoft.com/en-us/library/aa446519.aspx
- For more information on ASP.NET Mobile, see http://www.asp.net/mobile/road-map/
- For more information on adding ASP.NET Mobile source code support into Visual Studio 2008, see http://blogs.msdn.com/webdevtools/archive/2007/09/17/tip-trick-asp-net-mobile-development-with-visual-studio-2008.aspx
- For more information on security model permissions in Windows Mobile 6.x, see http://blogs.msdn.com/jasonlan/archive/2007/03/13/new-whitepaper-security-model-for-windows-mobile-5-0-and-windows-mobile-6.aspx

Chapter 2 – Architecture and Design Guidelines

Objectives

- Understand the fundamental concepts of software architecture.
- Learn the key design principles for software architecture.
- Learn the guidelines for key areas of software architecture.

Overview

Software architecture is often described as the organization or structure of a system, while the system represents a collection of components that accomplish a specific function or set of functions. In other words, architecture is focused on organizing components to support specific functionality. This organization of functionality is often referred to as grouping components into "areas of concern." Figure 1 illustrates common application architecture with components grouped by different areas of concern.

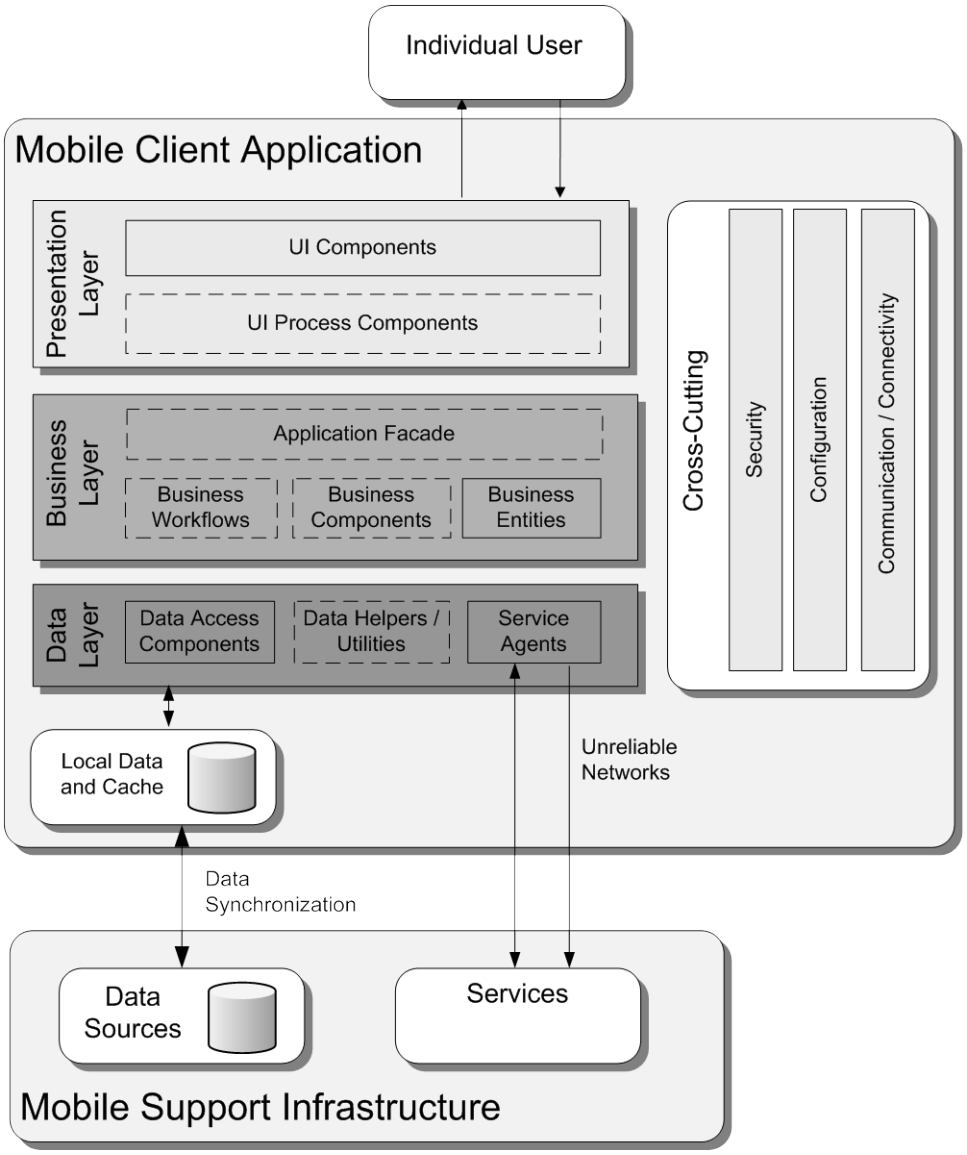

Figure 1 Common mobile application architecture

In addition to the grouping of components, other areas of concern focus on interaction between the components and how different components work together. The guidelines in this chapter examine different areas of concern that you should consider when designing the architecture of your application.

Key Design Principles

When getting started with your design, bear in mind the key principles that will help you to create architecture that meets "best practices," minimizes costs and maintenance requirements, and promotes usability and extendibility. The key principles are:

- **Separation of concerns**. Break your application into distinct features that overlap in functionality as little as possible.

- **Single Responsibility Principle**. Each component or a module should be responsible for only a specific feature or functionality.
- **Principle of least knowledge**. A component or an object should not know about internal details of other components or objects. Also known as the Law of Demeter (LoD).
- **Don't Repeat Yourself (DRY)**. There should be only one component providing a specific functionality; the functionality should not be duplicated in any other component.
- **Avoid doing a big design upfront**. If your application requirements are unclear, or if there is a possibility of the design evolving over time, avoid making a large design effort prematurely. This design principle is often abbreviated as BDUF.
- **Prefer composition over inheritance**. Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes.

Design Considerations

When designing an application or system, the goal of a software architect is to minimize the complexity by separating the design into different areas of concern. For example, the user interface (UI), business processing, and data access all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not mix code from other areas of concern. For example, UI processing components should not include code that directly accesses a data source, but instead should use either business components or data access components to retrieve data.

Follow these guidelines when designing an application:

- **Avoid doing all your design upfront**. If you are not clear with requirements or if there is the possibility of design evolution, it might be a good idea not to do complete design upfront. Instead, evolve the design as you progress through the project.
- **Separate the areas of concern**. Break your application into distinct features that overlap in functionality as little as possible. The main benefit of this approach is that a feature or functionality can be optimized independently of other features or functionality. Also, if one feature fails, it will not cause other features to fail as well, and they can run independently of one another. This approach also helps to make the application easier to understand and design, and facilitates management of complex interdependent systems.
- **Each component or module should have a single responsibility**. Each component or module should be responsible for only one specific feature or functionality. This makes your components cohesive and makes it easier to optimize the components if a specific feature or functionality changes.
- **A component or an object should not rely on internal details of other components or objects**. Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if needed, route it to appropriate subcomponents or other components. This helps in developing an application that is more maintainable and adaptable.
- **Do not duplicate functionality within an application**. There should be only one component providing a specific functionality—this functionality should not be duplicated in any other

component. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.

- **Identify the kinds of components you will need in your application**. The best way to do this is to identify patterns that match your scenario and then examine the types of components that are used by the pattern or patterns that match your scenario. For example, a smaller application may not need business workflow or UI processing components.
- **Group different types of components into logical layers**. Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers.
- **Keep design patterns consistent within each layer**. Within a logical layer, the design of components should be consistent for a particular operation. For example, if you choose to use the Table Data Gateway pattern to create an object that acts as a gateway to tables or views in a database, you should not include another pattern such as Repository, which uses a different paradigm for accessing data and initializing business entities.
- **Do not mix different types of components in the same logical layer**. For example, the UI layer should not contain business-processing components, but instead should contain components used to handle user input and process user requests.
- **Determine the type of layering you want to enforce**. In a strict layering system, components in layer A cannot call components in layer C; they always call components in layer B. In a more relaxed layering system, components in a layer can call components in other layers that are not immediately below it. In all cases, you should avoid upstream calls and dependencies.
- **Use abstraction to implement loose coupling between layers**. This can be accomplished by defining interface components such as a façade with well-known inputs and outputs that translate requests into a format understood by components within the layer. In addition, you can also use **Interface** types or abstract base classes to define a common interface or shared abstraction (dependency inversion) that must be implemented by interface components.
- **Do not overload the functionality of a component**. For example, a UI processing component should not contain data access code. A common anti-pattern named Blob is often found with base classes that attempt to provide too much functionality. A Blob object will often have hundreds of functions and properties providing business functionality mixed with cross-cutting functionality such as logging and exception handling. The large size is caused by trying to handle different variations of child functionality requirements, which requires complex initialization. The end result is a design that is very error-prone and difficult to maintain.
- **Understand how components will communicate with each other**. This requires an understanding of the deployment scenarios your application will need to support. You need to determine if communication across physical boundaries or process boundaries should be supported, or if all components will run within the same process.
- **Prefer composition over inheritance**. Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent

and child classes, thereby limiting the reuse of child classes. This also reduces the inheritance hierarchies, which can become very difficult to deal with.

- **Keep the data format consistent within a layer or component**. Mixing data formats will make the application more difficult to implement, extend, and maintain. Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation.
- **Keep cross-cutting code abstracted from the application business logic as much as possible**. Cross-cutting code refers to code related to security, communications, or operational management such as logging and instrumentation. Attempting to mix this code with business logic can lead to a design that is difficult to extend and maintain. Changes to the cross-cutting code would require touching all of the business logic code that is mixed with the cross-cutting code. Consider using frameworks that can help to implement the cross-cutting concerns.
- **Be consistent in the naming conventions used**. Check to see if naming standards have been established by the organization. If not, you should establish common standards that will be used for naming. This provides a consistent model that makes it easier for team members to evaluate code they did not write, which leads to better maintainability.
- **Establish the standards that should be used for exception handling**. For example, you should always catch exceptions at layer boundaries, you should not catch exceptions within a layer unless you can handle them in that layer, and you should not use exceptions to implement business logic. The standards should also include policies for error notification, logging, and instrumentation when there is an exception.

Architecture Frame

The following table lists the key areas to consider as you develop your architecture. Refer to the key issues in the table to understand where mistakes are most often made. The sections following this table provide guidelines for each of these areas.

Table 1 Architecture Frame

Area	Key issues
Authentication	• Lack of authorization across trust boundaries • Granular or improper authorization
Caching	• Caching volatile data not needed in an offline scenario • Caching sensitive data • Incorrect choice of caching store
Communication	• Incorrect choice of transport protocol • Chatty communication across physical and process boundaries • Failure to protect sensitive data
Concurrency and Transactions	• Not protecting concurrent access to static data • Deadlocks caused by improper locking • Not choosing the correct data concurrency model • Long-running transactions that hold locks on data • Using exclusive locks when not required

Area	Key issues
Configuration Management	• Lack of or incorrect configuration information • Not securing sensitive configuration information • Unrestricted access to configuration information
Coupling and Cohesion	• Incorrect grouping of functionality • No clear separation of concerns • Tight coupling across layers
Data Access	• Chatty calls to the database • Business logic mixed with data access code
Exception Management	• Failing to an unstable state • Revealing sensitive information to the end user • Using exceptions to control application flow • Not logging sufficient details about the exception
Layering	• Incorrect grouping of components within a layer • Not following layering and dependency rules
Logging and Instrumentation	• Lack of logging and instrumentation • Logging and instrumentation that is too fine-grained • Not making logging and instrumentation an option that is configurable at run time • Not suppressing and handling logging failures • Not logging business-critical functionality
State Management	• Using an incorrect state store • Not considering serialization requirements • Not persisting state when required
User Experience	• Not following published guidelines • Not considering accessibility • Creating overloaded interfaces with unrelated functionality
Validation	• Lack of validation across trust boundaries • Failure to validate for range, type, format, and length • Not reusing validation logic
Workflow	• Not considering management requirements • Choosing an incorrect workflow pattern • Not considering exception states and how to handle them

Authentication

Designing a good authentication strategy is important for the security and reliability of your application. Failure to design and implement a good authentication strategy can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attacks.

Consider the following guidelines when designing an authentication strategy:

- Identify your trust boundaries and authenticate users and calls across the trust boundaries.
- Do not store passwords in a database or data store as plain text. Instead, store a hash of the password.

- Enforce the use of strong passwords or password phrases.
- Do not transmit passwords over the wire in plain text.

Caching

Caching improves the performance and responsiveness of your application. However, a poorly designed caching strategy can degrade performance and responsiveness. You should use caching to optimize reference data lookups, avoid network round trips, and avoid unnecessary and duplicate processing. To implement caching, you must decide when to load the cache data. Try to load the cache asynchronously or by using a batch process to avoid client delays.

Consider following guidelines when designing a caching strategy:

- Do not cache volatile data unless that data is needed by a mobile device in an offline scenario
- Consider using ready-to-use cache data when working with an in-memory cache. For example, use a specific object instead of caching raw database data.
- Do not cache sensitive data unless you encrypt it.
- Do not depend on data still being in your cache. It may have been removed.

Communication

Communication concerns the interaction between components across different boundary layers. The mechanism you choose depends on the deployment scenarios your application must support. When crossing physical boundaries, you should use message-based communication. When crossing logical boundaries, you should use object-based communication.

Consider the following guidelines when designing communication mechanisms:

- To reduce round trips and improve communication performance, design chunky interfaces that communicate less often but with more information in each communication.
- Consider using message-based communication when crossing process or physical boundaries.
- If your messages don’t need to be received in exact order and don’t have dependencies on each other, consider using asynchronous communication to unblock processing or UI threads.

Concurrency and Transactions

When designing for concurrency and transactions related to accessing a database, it is important to identify the concurrency model you want to use and determine how transactions will be managed. For database concurrency, you can choose between an optimistic model, where the last update applied is valid, or a pessimistic model, where updates can only be applied to the latest version. Transactions can be executed within the database, or they can be executed in the business layer of an application. Where you choose to implement transactions depends on your transactional requirements. Concurrency should also be considered when accessing static data within the application or when using threads to perform asynchronous

operations. Static data is not thread-safe, which means that changes made in one thread will affect other threads using the same data.

Consider the following guidelines when designing for concurrency and transactions:
- If you have business-critical operations, consider wrapping them in transactions.
- Use connection-based transactions when accessing a single data source.
- Where you cannot use transactions, implement compensating methods to revert the data store to its previous state.
- Avoid holding locks for long periods; for example, when using long-running atomic transactions.
- Updates to shared data should be mutually exclusive, which is accomplished by applying locks or by using thread synchronization. This will prevent two threads from attempting to update shared data at the same time.
- Use synchronization support provided by collections when working with static or shared collections.

Configuration Management

Designing a good configuration-management mechanism is important for the security and flexibility of your application. Failure to do so can make your application vulnerable to a variety of attacks, and also leads to an administrative overhead for your application.

Consider the following guidelines when designing for configuration management:
- Consider using least-privileged process and service accounts.
- Encrypt sensitive information in your configuration store.
- Restrict access to your configuration information.
- Provide a separate administrative UI for editing configuration information.

Coupling and Cohesion

When designing components for your application, you should ensure that these components are highly cohesive, and that loose coupling is used across layers. *Coupling* is concerned with dependencies and functionality. When one component is dependent upon another component, it is tightly coupled to that component. Functionality can be decoupled by separating different operations into unique components. *Cohesion* concerns the functionality provided by a component. For example, a component that provides operations for validation, logging, and data access represents a component with very low cohesion. A component that provides operations for logging only represents high cohesion.

Consider the following guidelines when designing for coupling and cohesion:
- Partition application functionality into logical layers.
- Design for loose coupling between layers. Consider using abstraction to implement loose coupling between layers with interface components, common interface definitions, or shared abstraction. *Shared abstraction* is where concrete components depend on

abstractions and not on other concrete components (the principle of dependency inversion).
- Design for high cohesion. Components should contain only functionality that is specifically related to that component.
- Know the benefits and overhead of loosely coupled interfaces. While loose coupling requires more code, the benefits include a shortened dependency chain and a simplified build process.

Data Access

Designing an application to use a separate data access layer is important for maintainability and extensibility. The data access layer should be responsible for managing connections with the data source and for executing commands against the data source. Depending on your business entity design, the data access layer may have a dependency on business entities; however, the data access layer should never be aware of business processes or workflow components.

Consider the following guidelines when designing data access components:
- Avoid coupling your application model directly to the database schema. Instead, you should consider using an abstraction or mapping layer between the application model and database schema.
- If you are using SQL Server CE open the connection when application is launched and maintain it until the application is done running.
- Enforce data integrity in the database, not through data layer code.
- Move code that makes business decisions to the business layer.
- Avoid accessing the database directly from different layers in your application. Instead, all database interaction should be done through a data access layer.

Exception Management

Designing a good exception-management strategy is important for the security and reliability of your application. Failure to do so can make your application vulnerable to Denial of Service (DoS) attacks, and may also reveal sensitive and critical information. Raising and handling exceptions is an expensive process. It is important that the design also takes into account the performance considerations. A good approach is to design a centralized exception-management and logging mechanism, and to consider providing access points within your exception-management system to support instrumentation and centralized monitoring that assists system administrators.

Consider the following guidelines when designing an exception-management strategy:
- Do not catch exceptions unless you can handle them or need to add more information.
- Do not reveal sensitive information in exception messages and log files.
- Design an appropriate exception propagation strategy.
- Design a strategy for dealing with unhandled exceptions.
- Design an appropriate logging and notification strategy for critical errors and exceptions.

Layering

The use of layers in a design allows you to separate functionality into different areas of concern. In other words, layers represent the logical grouping of components within the design. You should also define guidelines for communication between layers. For example, layer A can access layer B, but layer B cannot access layer A.

Consider the following guidelines when designing layers:

- Layers should represent a logical grouping of components. For example, use separate layers for UI, business logic, and data access components.
- Components within a layer should be cohesive. In other words, the business layer components should provide only operations related to application business logic.
- Consider using an **Interface** type to define the interface for each layer. This will allow you to create different implementations of that interface to improve testability.
- For Web applications, implement a message-based interface between the presentation and business layers, even when the layers are not separated by a physical boundary. A message-based interface is better suited to stateless Web operations, provides a façade to the business layer, and allows you to physically decouple the business tier from the presentation tier if this is required by security policies or in response to a security audit.

Logging and Instrumentation

Designing a good logging and instrumentation strategy is important for the security and reliability of your application. Failure to do so can make your application vulnerable to repudiation threats, where users deny their actions. Log files may be required for legal proceedings to prove the wrongdoing of individuals. You should audit and log activity across the layers of your application. Using logs, you can detect suspicious activity, which can provide early indication of a serious attack. Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access, and by the same routines that access the resource. Instrumentation can be implemented by using performance counters and events to give administrators information about the state, performance, and health of an application.

Consider the following guidelines when designing a logging and instrumentation strategy:

- Centralize your logging and instrumentation mechanism.
- Design instrumentation within your application to detect system- and business-critical events.
- Consider how you will access and pass auditing and logging data across application layers.
- Create secure log file management policies. Protect log files from unauthorized viewing.
- Do not store sensitive information in the log files.
- Consider allowing your log sinks, or trace listeners, to be configurable so that they can be modified at run time to meet deployment environment requirements.

State Management

State management concerns the persistence of data that represents the state of a component, operation, or step in a process. State data can be persisted by using different formats and stores. The design of a state-management mechanism can affect the performance of your application. You should only persist data that is required, and you must understand the options that are available for managing state.

Consider the following guidelines when designing a state management mechanism:

- Keep your state management as lean as possible; persist the minimum amount of data required to maintain state.
- Make sure that your state data is serializable if it needs to be persisted or shared across process and network boundaries.

User Experience

Designing for an effective user experience can be critical to the success of your application. If navigation is difficult, or users are directed to unexpected pages, the user experience can be negative.

Consider the following guidelines when designing for an effective user experience:

- Design for a consistent navigation experience. Use composite patterns for the look and feel, and controller patterns such as Model-View-Controller (MVC) for UI processing.
- Design the interface so that each page or section is focused on a specific task.
- Consider breaking large pages with a lot of functionality into smaller pages.
- Consider using published UI guidelines. In many cases, an organization will have published guidelines to which you should adhere.

Validation

Designing an effective validation mechanism is important for the security and reliability of your application. Failure to do so can make your application vulnerable to cross-site scripting, SQL injection, buffer overflow, and other types of malicious input attacks. However, there is no standard definition that can differentiate valid input from malicious input. In addition, how your application actually uses the input influences the risks associated with exploitation of the vulnerability.

Consider the following guidelines when designing a validation mechanism:

- Centralize your validation approach, if it can be reused.
- Constrain, reject, and sanitize user input. In other words, assume that all user input is malicious.
- Validate input data for length, format, type, and range.

Workflow

Workflow components are used when an application must execute a series of information-processing tasks that are dependent on the information content. The values that affect information-processing steps can be anything from data checked against business rules to human interaction and input. When designing workflow components, it is important to consider the options that are available for management of the workflow.

Consider the following guidelines when designing a workflow component:

- Determine management requirements. For example, if a business user needs to manage the workflow, you require a solution that provides an interface that the business user can understand.
- Determine how exceptions will be handled.
- Use service interfaces to interact with external workflow providers.
- If supported, use designers and metadata instead of code to define the workflow.
- With human workflow, consider the nondeterministic nature of users. In other words, you cannot determine when a task will be completed, or if it will be completed correctly.

Pattern Map

Key patterns are organized by the key categories detailed in the Architecture Frame in the following table. Consider using these patterns when making design decisions for each category

Table 2 Pattern Map

Category	Relevant patterns
Caching	• Cache Dependency
Communication	• Pipes and Filters • Service Interface
Concurrency and Transactions	• Optimistic Offline Lock • Pessimistic Offline Lock
Coupling and Cohesion	• Adapter • Dependency Injection
Data Access	• Active Record • Query Object • Row Data Gateway • Table Data Gateway
Layering	• Façade • Layered Architecture

- For more information on the Service Interface, and Layered Architecture patterns, see "Enterprise Solution Patterns Using Microsoft .NET" at http://msdn.microsoft.com/en-us/library/ms998469.aspx
- For more information on the Active Record, Data Mapper, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository, Row Data Gateway, and Table Data

Gateway patterns, see “Patterns of Enterprise Application Architecture (P of EAA)” at http://martinfowler.com/eaaCatalog/
- For more information on the Adapter and Façade patterns, see “data & object factory” at http://www.dofactory.com/Patterns/Patterns.aspx
- For more information on the Pipes and Filters pattern, see “Integration Patterns” at http://msdn.microsoft.com/en-us/library/ms978729.aspx

Pattern Descriptions

- **Active Record**. Include a data access object within a domain entity.
- **Adapter**. An object that supports a common interface and translates operations between the common interface and other objects that implement similar functionality with different interfaces.
- **Cache Dependency**. Use external information to determine the state of data stored in a cache.
- **Dependency Injection**. Use a base class or interface to define a shared abstraction that can be used to inject object instances into components that interact with the shared abstraction interface.
- **Façade**. Implement a unified interface to a set of operations to provide a simplified, reduced coupling between systems.
- **Optimistic Offline Lock**. Ensure that changes made by one session do not conflict with changes made by another session.
- **Pessimistic Offline Lock**. Prevent conflicts by forcing a transaction to obtain a lock on data before using it.
- **Pipes and Filters**. Route messages through pipes and filters that can modify or examine the message as it passes through the pipe.
- **Query Object**. An object that represents a database query.
- **Row Data Gateway**. An object that acts as a gateway to a single record in a data source.
- **Service Interface**. A programmatic interface that other systems can use to interact with the service.
- **Table Data Gateway**. An object that acts as a gateway to a table in a data source.

Additional Resources

- For more information, see *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn.microsoft.com/en-us/library/ms998469.aspx.
- For more information, see *Integration Patterns* at http://msdn.microsoft.com/en-us/library/ms978729.aspx.
- For more information, see *Cohesion and Coupling* at http://msdn.microsoft.com/en-us/magazine/cc947917.aspx.
- For more information on caching, see *Caching Architecture Guide for .NET Framework Applications* at http://msdn.microsoft.com/en-us/library/ms978498.aspx.

- For more information on exception management, see *Exception Management Architecture Guide* at http://msdn.microsoft.com/en-us/library/ms954599.aspx.

Chapter 3 – Presentation Layer Guidelines

Objectives

- Understand how the presentation layer fits into typical application architecture.
- Understand the components of the presentation layer.
- Learn the steps for designing the presentation layer.
- Learn the common issues faced while designing the presentation layer.
- Learn the key guidelines for designing the presentation layer.
- Learn the key patterns and technology considerations for designing the presentation layer.

Overview

The presentation layer contains the components that implement and display the user interface and manage user interaction. This layer includes controls for user input and display, in addition to components that organize user interaction. Figure 1 shows how the presentation layer fits into a common application architecture.

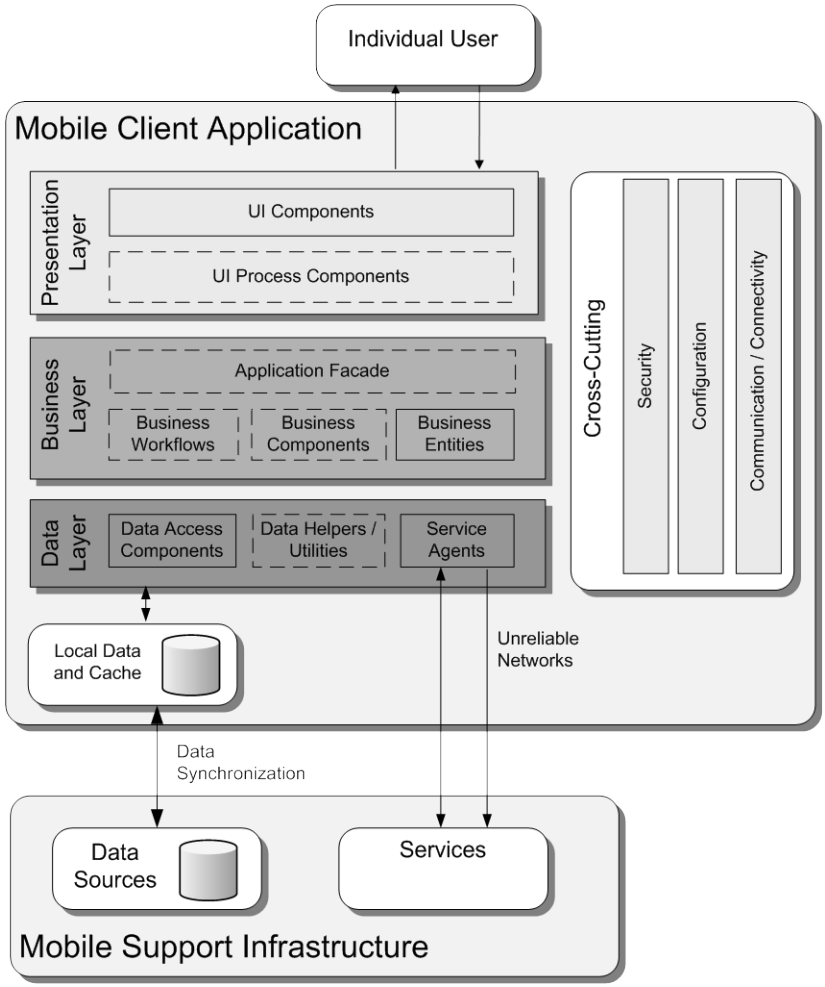

Figure 1 A typical application showing the presentation layer and the components it may contain

Presentation Layer Components

- **User interface (UI) components**. User interface components provide a way for users to interact with the application. They render and format data for users. They also acquire and validate data input by the user.
- **User process components**. User process components synchronize and orchestrate user interactions. Separate user process components may be useful if you have a complicated UI. Implementing common user interaction patterns as separate user process components allows you to reuse them in multiple UIs.

Approach

The following steps describe the process you should adopt when designing the presentation layer for your application. This approach will ensure that you consider all of the relevant factors as you develop your architecture:

1. **Identify your client type**. Choose a client type that satisfies your requirements and adheres to the infrastructure and deployment constraints of your organization. If your mobile devices users will be intermittently connected to the network, a mobile rich client is probably your best choice.
2. **Determine how you will present data**. Choose the data format for your presentation layer and decide how you will present the data in your UI.
3. **Determine your data-validation strategy**. Use data-validation techniques to protect your system from untrusted input.
4. **Determine your business logic strategy**. Factor out your business logic to decouple it from your presentation layer code.
5. **Determine your strategy for communication with other layers**. If your application has multiple layers, such as a data access layer and a business layer, determine a strategy for communication between your presentation layer and other layers.

Design Considerations

There are several key factors that you should consider when designing your presentation layer. Use the following principles to ensure that your design meets the requirements for your application, and follows best practices:

- **Choose the appropriate UI technology.** Determine if you will implement a rich (smart) client, a Web client, or a rich Internet application (RIA). Base your decision on application requirements, and on organizational and infrastructure constraints.
- **Use the relevant patterns.** Review the presentation layer patterns for proven solutions to common presentation problems.
- **Design for separation of concerns.** Use dedicated UI components that focus on rendering and display. Use dedicated UI process components to manage the processing of user interaction.
- **Consider human interface guidelines.** Review your organization’s guidelines for UI design. Review established UI guidelines based on the client type and technologies that you have chosen.

- **Adhere to user-driven design principles.** Before designing your presentation layer, understand your customer. Use surveys, usability studies, and interviews to determine the best presentation design to meet your customer's requirements.

Presentation Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Category	Common issues
Caching	• Caching volatile data. • Caching unencrypted sensitive data. • Incorrect choice of caching store. • Assuming that data will still be available in the cache – it may have expired and been removed.
Exception Management	• Failing to catch unhandled exceptions. • Failing to clean up resources and state after an exception occurs. • Revealing sensitive information to the end user. • Using exceptions to control application flow. • Catching exceptions you do not handle. • Using custom exceptions when not necessary.
Input	• Failing to design for intuitive use, or implementing overly complex interfaces. • Failing to design for accessibility. • Failing to design for different screen sizes and resolutions. • Failing to design for different device and input types, such as touch-screen, and pen and ink–enabled devices.
Layout	• Failing to consider different screen resolutions of mobile devices. • Implementing an overly complex layout. • Failing to choose appropriate layout components and technologies. • Failing to adhere to accessibility and usability guidelines and standards. • Failing to support localization and globalization.
Navigation	• Inconsistent navigation. • Duplication of logic to handle navigation events. • Failing to use Star forms style of navigation. • Failing to manage state with wizard navigation.
Request Processing	• Blocking the UI during long-running requests. • Mixing processing and rendering logic. • Choosing an inappropriate request-handling pattern.

Category	Common issues
User Experience	• Displaying unhelpful error messages. • Lack of responsiveness. • Overly complex user interfaces. • Lack of user personalization. • Lack of user empowerment. • Designing inefficient user interfaces.
UI Components	• Creating custom components that are not necessary. • Failing to maintain state in the Model-View-Controller (MVC) pattern. • Choosing inappropriate UI components.
UI Process Components	• Implementing UI process components when not necessary. • Implementing the wrong design patterns. • Mixing business logic with UI process logic. • Mixing rendering logic with UI process logic.
Validation	• Failing to validate all input. • Failing to correctly handle validation errors. • Not identifying business rules that are appropriate for validation. • Failing to log validation failures.

Caching

Caching is one of the best mechanisms you can use to improve application performance and UI responsiveness. Use data caching to optimize data lookups and avoid network round trips. Cache the results of expensive or repetitive processes to avoid unnecessary duplicate processing.

Consider the following guidelines when designing your caching strategy:

- Consider the limited resources on mobile device while deciding caching strategy.
- Do not cache volatile data.
- Consider using ready-to-use cache data when working with an in-memory cache. For example, use a specific object instead of caching raw database data.
- Do not cache sensitive data unless you encrypt it.
- Do not depend on data still being in your cache. It may have been removed.

Exception Management

Design a centralized exception-management mechanism for your application that catches and throws exceptions consistently. Pay particular attention to exceptions that propagate across layer or tier boundaries, as well as exceptions that cross trust boundaries. Design for unhandled exceptions so they do not impact application reliability or expose sensitive information.

Consider the following guidelines when designing your exception management strategy:

- Use user-friendly error messages to notify users of errors in the application.
- Avoid exposing sensitive data in error pages, error messages, log files, and audit files.

- Design a global exception handler that displays a global error page or an error message for all unhandled exceptions.
- Differentiate between system exceptions and business errors. In the case of business errors, display a user-friendly error message and allow the user to retry the operation. In the case of system exceptions, check to see if the exception was caused by issues such as system or database failure, display a user-friendly error message, and log the error message, which will help in troubleshooting.
- Avoid using exceptions to control application logic.

Input

Design a user input strategy based on your application input requirements. For maximum usability, follow the established guidelines defined in your organization, and the many established industry usability guidelines based on years of user research into input design and mechanisms.

Consider the following guidelines when designing your input collection strategy:

- Use forms-based input controls for normal data-collection tasks.
- Use a document-based input mechanism for collecting input in Microsoft Office–style documents.
- Implement a wizard-based approach for more complex data collection tasks, or for input that requires a workflow.
- Design to support localization by avoiding hard-coded strings and using external resources for text and layout.
- Consider accessibility in your design. You should consider users with disabilities when designing your input strategy; for example, implement text-to-speech software for blind users, or enlarge text and images for users with poor sight. Support keyboard-only scenarios where possible for users who cannot manipulate a pointing device.

Layout

Design your UI layout so that the layout mechanism itself is separate from the individual UI components and UI process components. When choosing a layout strategy, consider whether you will have a separate team of designers building the layout, or whether the development team will create the UI. If designers will be creating the UI, choose a layout approach that does not require code or the use of development-focused tools.

As mobile devices have smaller screen size and limited input it might be difficult to incorporate all of an applications' functionality into a single screen. You may consider a design pattern to break up information and actions across several screens. While doing this, you also need to keep mobile applications and their interfaces as concise as possible, as having a large number of screens may be difficult and frustrating for the user to easily navigate within the limited framework of the mobile environment.

Consider the following guidelines when designing your layout strategy:

- Use templates to provide a common look and feel to all of the UI screens.
- Use a common look and feel for all elements of your UI to maximize accessibility and ease of use.
- Consider device-dependent input, such as touch screens, ink, or speech, in your layout. For example, with touch-screen input you will typically use larger buttons with more spacing between them than you would with mouse or keyboard inputs.
- Use well known design patterns, to separate the layout design from interface processing.

Navigation

Design your navigation strategy so that users can navigate easily through your screens or pages, and so that you can separate navigation from presentation and UI processing. Ensure that you display navigation links and controls in a consistent way throughout your application to reduce user confusion and hide application complexity.

Consider the following guidelines when designing your navigation strategy:

- Use well-known design patterns to decouple the UI from the navigation logic where this logic is complex. For mobile, use the “star forms” concept where you have one central screen that acts as a “home page”. From that screen cascade to other screens to perform more specific functions, always returning to the central home page screen.
- While designing toolbars and menus to help users find functionality provided by the UI, consider the smaller screen size of devices.
- Consider using wizards to implement navigation between forms in a predictable way.
- Determine how you will preserve navigation state if the application must preserve this state between sessions.
- Consider using the Command Pattern to handle common actions from multiple sources.

Request Processing

Design your request processing with user responsiveness in mind, as well as code maintainability and testability.

Consider the following guidelines when designing request processing:

- Use asynchronous operations or worker threads to avoid blocking the UI for long-running actions.
- Avoid mixing your UI processing and rendering logic.

User Experience

Good user experience can make the difference between a usable and unusable application. Carry out usability studies, surveys, and interviews to understand what users require and expect from your application, and design with these results in mind.

Consider the following guidelines when designing for user experience:

- When developing a rich Internet application (RIA), avoid synchronous processing where possible.
- Do not design overloaded or overly complex interfaces. Provide a clear path through the application for each key user scenario.
- Design to support user personalization, localization, and accessibility.
- Design for user empowerment. Allow the user to control how he or she interacts with the application, and how it displays data to them.

UI Components

UI components are the controls and components used to display information to the user and accept user input. Be careful not to create custom controls unless it is necessary for specialized display or data collection.

Consider the following guidelines when designing UI components:

- Take advantage of the data-binding features of the controls you use in the UI.
- Create custom controls or use third-party controls only for specialized display and data-collection tasks.
- When creating custom controls, extend existing controls if possible instead of creating a new control.
- Consider implementing designer support for custom controls to make it easier to develop with them.
- Consider maintaining the state of controls as the user interacts with the application instead of reloading controls with each action.

UI Process Components

UI process components synchronize and orchestrate user interactions. UI processing components are not always necessary; create them only if you need to perform significant processing in the presentation layer that must be separated from the UI controls. Be careful not to mix business and display logic within the process components; they should be focused on organizing user interactions with your UI.

Consider the following guidelines when designing UI processing components:

- Do not create UI process components unless you need them.
- If your UI requires complex processing or needs to talk to other layers, use UI process components to decouple this processing from the UI.
- Consider dividing UI processing into three distinct roles: Model, View, and Controller/Presenter, by using the MVC or MVP pattern.
- Avoid business rules, with the exception of input and data validation, in UI processing components.
- Consider using abstraction patterns, such as dependency inversion, when UI processing behavior needs to change based on the run-time environment.

- Where the UI requires complex workflow support, create separate workflow components that use a workflow system such as Windows Workflow or a custom mechanism.

Validation

Designing an effective input and data-validation strategy is critical to the security of your application. Determine the validation rules for user input as well as for business rules that exist in the presentation layer.

Consider the following guidelines when designing your input and data validation strategy:

- Validate all input data to reduce errors caused by invalid data.
- Design your validation strategy to constrain, reject, and sanitize malicious input.
- Use the built-in validation controls where possible, when working with .NET Compact Framework.

Pattern Map

Key patterns are organized by key categories, as detailed in the Presentation Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Category	Relevant patterns
Caching	• Cache Dependency
Exception Management	• Exception Shielding
Layout	• Template View
Navigation	• Command Pattern
Presentation Entities	• Entity Translator
User Experience	• Asynchronous Callback • Chain of Responsibility
UI Processing Components	• Model-View-Controller (MVC) • Presentation Model

- For more information on the Model-View-Controller (MVC), Template View patterns, see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/
- For more information on the Presentation Model patterns, see "Patterns in the Composite Application Library" at http://msdn.microsoft.com/en-us/library/cc707841.aspx
- For more information on the Chain of responsibility and Command pattern, see "data & object factory" at http://www.dofactory.com/Patterns/Patterns.aspx
- For more information on the Asynchronous Callback pattern, see "Creating a Simplified Asynchronous Call Pattern for Windows Forms Applications" at http://msdn.microsoft.com/en-us/library/ms996483.aspx
- For more information on the Exception Shielding and Entity Translator patterns, see "Useful Patterns for Services" at http://msdn.microsoft.com/en-us/library/cc304800.aspx

Pattern Descriptions

- **Asynchronous Callback.** Execute long-running tasks on a separate thread that executes in the background, and provide a function for the thread to call back into when the task is complete.
- **Cache Dependency.** Use external information to determine the state of data stored in a cache.
- **Chain of Responsibility.** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- **Command Pattern.** Encapsulate request processing in a separate command object with a common execution interface.
- **Entity Translator.** An object that transforms message data types into business types for requests, and reverses the transformation for responses.
- **Exception Shielding.** Prevent a service from exposing information about its internal implementation when an exception occurs.
- **Model-View-Controller**. Separate the UI code into three separate units: Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Page Controller**. Accept input from the request and handle it for a specific page or action on a Web site.
- **Presentation Model**. Move all view logic and state out of the view, and render the view through data-binding and templates.
- **Template View**. Implement a common template view, and derive or construct views using this template view.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology. The guidelines also contain suggestions for common patterns that are useful for specific types of application and technology.

Mobile Applications

Consider the following guidelines when designing a mobile application:

- If you want to build full-featured connected, occasionally connected, and disconnected executable applications that run on a wide range of Microsoft Windows®–based devices, consider using the Microsoft Windows Compact Framework.
- If you want to build applications that support rich media and interactivity, consider using Microsoft Silverlight® for Mobile.

Additional Resources

- *For more information, see Microsoft Inductive User Interface Guidelines* at http://msdn.microsoft.com/en-us/library/ms997506.aspx.

- *For more information, see User Interface Control Guidelines* at http://msdn.microsoft.com/en-us/library/bb158625.aspx.
- *For more information, see User Interface Text Guidelines* at http://msdn.microsoft.com/en-us/library/bb158574.aspx.

Chapter 4: Business Layer Guidelines

Objectives

- Understand how the business layer fits into the overall application architecture.
- Understand the components of the business layer.
- Learn the steps for designing these components.
- Learn about the common issues faced when designing the business layer.
- Learn the key guidelines for designing the business layer.
- Learn the key patterns and technology considerations.

Overview

This chapter describes the design process for business layers, and contains key guidelines that cover the important aspects you should consider when designing business layers and business components. These guidelines are organized into categories that include designing business layers and implementing appropriate functionality such as security, caching, exception management, logging, and validation. These categories represent the key areas where mistakes occur most often in business layer design. Figure 1 shows how the business layer fits into typical application architecture.

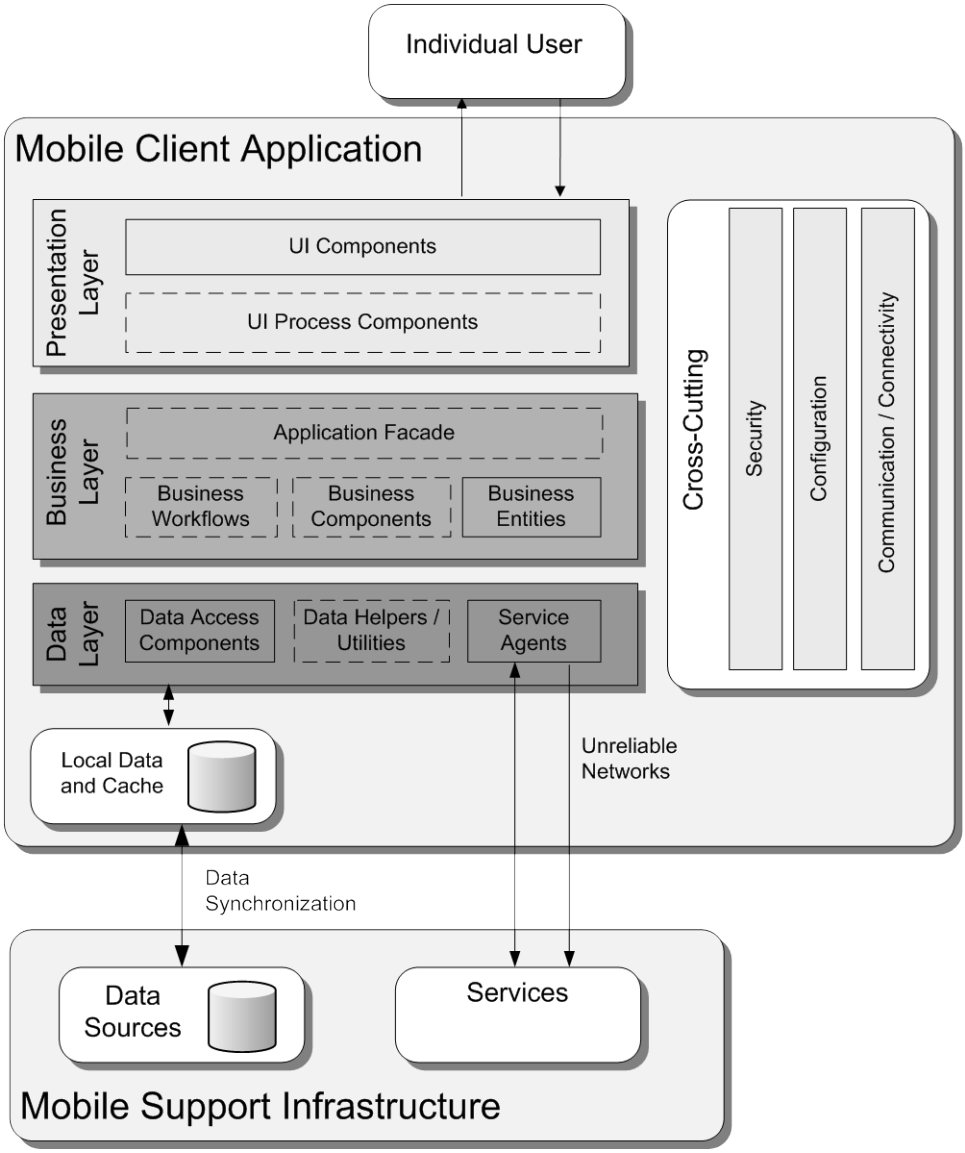

Figure 1 A typical application showing the business layer and the components it may contain

Key Business Components

The following list explains the roles and responsibilities of the main components within the business layer:

- **Application façade** (optional). An application façade combines multiple business operations into a single message-based operation. You might access the application façade from the presentation layer by using different communication technologies.
- **Business components**. Within the business layer there are different components that provide business services, such as processing business rules and interacting with data access components. For example, you might have a business component that implements the transaction script pattern, which allows you to execute multiple operations within a single component used to manage the transaction. Another business component might be used to process requests and apply business rules.

- **Business entities**. Business components used to pass data between other components are considered business entities. The data can represent real-world business entities, such as products and orders, or database entities, such as tables and views. Consider using scalar values as business entities. Alternatively, business entities can be implemented using data structures such as DataSets and Extensible Markup Language (XML) documents.
- **Business workflows**. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. Business workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.

Approach

When designing a business layer, you must also take into account the design requirements for the main constituents of the layer, such as business components, business entities, and business workflow components. This section briefly explains the main activities involved in designing each of the components and the business layer itself. Perform the following key activities in each of these areas when designing your data layer:

1. **Create an overall design for your business layer:**
    - Identify the consumers of your business layer.
    - Determine how you will expose your business layer.
    - Determine the security requirements for your business layer.
    - Determine the validation requirements and strategy for your business layer.
    - Determine the caching strategy for your business layer.
    - Determine the exception-management strategy for your business layer.
2. **Design your business components**:
    - Identify business components your application will use.
    - Make key decisions about location, coupling, and interactions for business components.
    - Choose appropriate transaction support.
    - Identify how your business rules are handled.
    - Identify patterns that fit the requirements.
3. **Design your business entity components**:
    - Identify common data formats for the business entities.
    - Choose the data format.
    - Optionally, choose a design for your custom objects.
    - Optionally, determine how you will serialize the components.
4. **Design your workflow components:**
    - Identify workflow style using scenarios.
    - Choose an authoring mode.
    - Determine how rules will be handled.
    - Choose a workflow solution.
    - Design business components to support workflow.

Design Considerations

When designing a business layer, the goal of a software architect is to minimize the complexity by separating tasks into different areas of concern. For example, business processing, business workflow, and business entities all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not include code related to other areas of concern.

Consider the following guidelines when designing the business layer:

- **Decide if you need a separate business layer**. It is always a good idea to use a separate business layer where possible to improve the maintainability of your application.
- **Identify the responsibilities of your business layer**. Use a business layer for processing complex business rules, transforming data, applying policies, and for validation.
- **Do not mix different types of components in your business layer**. Use a business layer to decouple business logic from presentation and data access code, and to simplify the testing of business logic.
- **Reuse common business logic**. Use a business layer to centralize common business logic functions and promote reuse.
- **Avoid tight coupling between layers**. Use abstraction when creating an interface for the business layer. The abstraction can be implemented using public object interfaces, common interface definitions, abstract base classes, or messaging.

Business Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Category	Common issues
Business Components	• Overloading business components, by mixing unrelated functionality. • Mixing data access logic within business logic in business components. • Not considering the use of message-based interfaces to expose business components.
Business Entities	• Choosing incorrect data formats for your business entities. • Not considering serialization requirements.
Caching	• Caching volatile data not required in an offline scenario • Caching too much data in the business layer. • Failing to cache data in a ready-to-use format. • Caching sensitive data in unencrypted form.

Category	Common issues
Coupling and Cohesion	• Tight coupling across layers. • No clear separation of concerns within the business layer. • Failing to use a message-based interface between layers.
Concurrency and Transactions	• Not preventing concurrent access to static data that is not read-only. • Not choosing the correct data concurrency model. • Using long-running transactions that hold locks on data.
Data Access	• Accessing the database directly from the business layer. • Mixing data access logic within business logic in business components.
Exception Management	• Revealing sensitive information to the end user. • Using exceptions to control application flow. • Not logging sufficient detail from exceptions. • Failing to appropriately notify users with useful error messages.
Logging and Instrumentation	• Failing to add adequate instrumentation to business components. • Failing to log system-critical and business-critical events. • Not suppressing logging failures.
Validation	• Relying on validation that occurs in the presentation layer. • Failure to validate for length, range, format and type. • Not reusing the validation logic.
Workflows	• Not considering application management requirements. • Choosing an incorrect workflow pattern. • Not considering how to handle all exception states. • Choosing an incorrect workflow technology.

Business Components

Business components implement business rules in diverse patterns, and accept and return simple or complex data structures. Your business components should expose functionality in a way that is agnostic to the data stores and services required to perform the work. Compose your business components in meaningful and transactionally consistent ways. Designing business components is an important task. If you fail to design business components correctly, the result is likely to be code that is impossible to maintain.

Consider the following guidelines when designing business components:

- Avoid mixing data access logic and business logic within your business components.
- Design components to be highly cohesive. In other words, you should not overload business components by adding unrelated or mixed functionality.

- If you want to keep business rules separate from business data, consider using business process components to implement your business rules.
- If your application has volatile business rules, store them in a rules engine.
- If the business process involves multiple steps and long-running transactions, consider using workflow components.

Business Entities

Business entities store data values and expose them through properties; they provide stateful programmatic access to the business data and related functionality. Therefore, designing or choosing appropriate business entities is vitally important for maximizing the performance and efficiency of your business layer.

Consider the following guidelines when designing business entities:

- Choose appropriate data formats for your business entities. For smaller data-driven applications consider using DataSets, and for document-centric data consider using XML for the data format. For other types of applications, consider using custom objects instead.
- If the tables in the database represent business entities, consider using the Table Module pattern.
- Consider the serialization requirements of your business entities. For example, if you are storing business entities in a central location for state management, or passing business entities across process or network boundaries, they will need to support serialization.
- Minimize the number of calls made across physical tiers. For example, use the Data Transfer Object (DTO) pattern.

Caching

Designing an appropriate caching strategy for your business layer is important for the performance and responsiveness of your application. Use caching to optimize reference data lookups, avoid network round trips, and avoid unnecessary and duplicated processing. As part of your caching strategy, you must decide when and how to load the cache data. To avoid client delays, load the cache asynchronously or by using a batch process.

Consider the following guidelines when designing a caching strategy:

- Consider caching static data that will be reused regularly within the business layer.
- Consider caching data that cannot be retrieved from the database quickly and efficiently.
- Consider caching data in a ready-to-use format within your business layer.
- Avoid caching sensitive data if possible, or design a mechanism to protect sensitive data in the cache.

Coupling and Cohesion

When designing components for your business layer, ensure that they are highly cohesive, and implement loose coupling between layers. This helps to improve the scalability of your application.

Consider the following guidelines when designing for coupling and cohesion:

- Avoid circular dependencies. The business layer should know only about the layer below (the data access layer), and not the layer above (the presentation layer or external applications that access the business layer directly).
- Use abstraction to implement a loosely coupled interface. This can be achieved with interface components, common interface definitions, or shared abstraction where concrete components depend on abstractions and not on other concrete components (the principle of Dependency Inversion).
- Design for tight coupling within the business layer unless dynamic behavior requires loose coupling.
- Design for high cohesion. Components should contain only functionality specifically related to that component.
- Avoid mixing data access logic with business logic in your business components.

Concurrency and Transactions

When designing for concurrency and transactions, it is important to identify the appropriate concurrency model and determine how you will manage transactions. You can choose between an optimistic model and a pessimistic model for concurrency. With *optimistic concurrency*, locks are not held on data and updates require code to check, usually against a timestamp, that the data has not changed since it was last retrieved. With *pessimistic concurrency*, data is locked and cannot be updated by another operation until the lock is released.

Consider the following guidelines when designing for concurrency and transactions:

- Consider transaction boundaries, so that retries and composition are possible.
- Where you cannot apply a commit or rollback, or if you use a long-running transaction, implement compensating methods to revert the data store to its previous state in case an operation within the transaction fails.
- Avoid holding locks for long periods; for example, when executing long-running atomic transactions or when locking access to shared data.
- Choose an appropriate transaction isolation level, which defines how and when changes become available to other operations.

Data Access

Designing an effective data-access strategy for your business layer is important for maximizing maintainability and the separation of concerns. Failing to do so can make your application difficult to manage and extend as business requirements change. An effective data-access strategy will allow your business layer to adapt to changes in the underlying data sources. It will also make it easier to reuse functionality and components in other applications.

Consider the following guidelines when designing a data-access strategy:

- Avoid mixing data-access code and business logic within your business components.
- Avoid directly accessing the database from your business layer.

- Consider using a separate data access layer for access to the database.

Exception Management

Designing an effective exception-management solution for your business layer is important for the security and reliability of your application. Failing to do so can leave your application vulnerable to Denial of Service (DoS) attacks, and may allow it to reveal sensitive and critical information about your application. Raising and handling exceptions is an expensive operation, so it is important that your exception management design takes into account the impact on performance.

When designing an exception-management strategy, consider following guidelines:

- Do not use exceptions to control business logic.
- Only catch exceptions that you can handle, or if you need to add information. For example, catch data conversion exceptions that can occur when trying to convert null values.
- Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer.
- Design an approach for catching and handling unhandled exceptions.
- Design an appropriate logging and notification strategy for critical errors and exceptions that does not reveal sensitive information.

Logging and Instrumentation

Designing a good logging and instrumentation solution for your business layer is important for the security and reliability of your application. Failing to do so can leave your application vulnerable to repudiation threats, where users deny their actions. Log files may also be required to prove wrongdoing in legal proceedings. Auditing is generally considered most authoritative if the log information is generated at the precise time of resource access, and by the same routine that accesses the resource. Instrumentation can be implemented using performance counters and events. System-monitoring tools can use this instrumentation, or other access points, to provide administrators with information about the state, performance, and health of an application.

Consider the following guidelines when designing a logging and instrumentation strategy:

- Centralize logging and instrumentation for your business layer.
- Include instrumentation for system-critical and business-critical events in your business components.
- Do not store business-sensitive information in the log files.
- Ensure that a logging failure does not affect normal business layer functionality.
- Consider auditing and logging all access to functions within business layer.

Validation

Designing an effective validation solution for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attacks. There is no comprehensive definition of what constitutes a valid input or malicious input. In addition, how your application uses input influences the risk of the exploit.

Consider the following guidelines when designing a validation strategy:

- Validate all input and method parameters within the business layer, even when input validation occurs in the presentation layer.
- Centralize your validation approach, if it can be reused.
- Constrain, reject, and sanitize user input. In other words, assume that all user input is malicious.
- Validate input data for length, range, format, and type.

Workflows

Workflow components are used only when your application must support a series of tasks that are dependent on the information being processed. This information can be anything from data checked against business rules to human interaction. When designing workflow components, it is important to consider how you will manage the workflows, and to understand the available options.

Consider the following guidelines when designing a workflow strategy:

- Implement workflows within components that involve a multi-step or long-running process.
- Choose an appropriate workflow style depending on the application scenario.
- Handle fault conditions within workflows, and expose suitable exceptions.
- If the component must execute a specified set of steps sequentially and synchronously, consider using the pipeline pattern.
- If the process steps can be executed asynchronously in any order, consider using the event pattern.

Pattern Map

Key patterns are organized by the key categories detailed in the Business Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Category	Relevant patterns
Business Components	• Application Façade • Chain of Responsibility • Command
Business Entities	• Entity Translator • Table Module

Concurrency and Transactions	• Capture Transaction Details • Coarse-Grained Lock • Implicit Lock • Optimistic Offline Lock • Pessimistic Offline Lock • Transaction Script
Data Access	• Active Record • Query Object • Row Data Gateway • Table Data Gateway
Workflows	• Data-driven workflow • Human workflow • Sequential workflow • State-driven workflow

- For more information on the Domain Model, Table Module, Coarse-Grained Lock, Implicit Lock, Transaction Script, Active Record, Data Mapper, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository, Row Data Gateway, and Table Data Gateway patterns, see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/
- For more information on the Façade, Chain of Responsibility, and Command patterns, see "data & object factory" at http://www.dofactory.com/Patterns/Patterns.aspx
- For more information on the Entity Translator pattern, see "Useful Patterns for Services" at http://msdn.microsoft.com/en-us/library/cc304800.aspx
- For more information on the Capture Transaction Details pattern, see "Data Patterns" at http://msdn.microsoft.com/en-us/library/ms998446.aspx
- For more information on the Data-Driven Workflow, Human Workflow, Sequential Workflow, and State-Driven Workflow, see "Windows Workflow Foundation Overview" at http://msdn.microsoft.com/en-us/library/ms734631.aspx and "Workflow Patterns" at http://www.workflowpatterns.com/

Pattern Descriptions

- **Active Record.** Include a data access object within a domain entity.
- **Capture Transaction Details**. Create database objects, such as triggers and shadow tables, to record changes to all tables belonging to the transaction.
- **Chain of Responsibility**. Avoid coupling the sender of a request to its receiver by allowing more than one object to handle the request.
- **Coarse Grained Lock**. Lock a set of related objects with a single lock.
- **Command**. Encapsulate request processing in a separate command object with a common execution interface.
- **Data-driven Workflow**. A workflow that contains tasks whose sequence is determined by the values of data in the workflow or the system.
- **Entity Translator**. An object that transforms message data types to business types for requests, and reverses the transformation for responses.

- **Human Workflow**. A workflow that involves tasks performed manually by humans.
- **Implicit Lock**. Use framework code to acquire locks on behalf of code that accesses shared resources.
- **Optimistic Offline Lock**. Ensure that changes made by one session do not conflict with changes made by another session.
- **Pessimistic Offline Lock**. Prevent conflicts by forcing a transaction to obtain a lock on data before using it.
- **Query Object**. An object that represents a database query.
- **Row Data Gateway**. An object that acts as a gateway to a single record in a data source.
- **Sequential Workflow**. A workflow that contains tasks that follow a sequence, where one task is initiated after completion of the preceding task.
- **State-driven Workflow**. A workflow that contains tasks whose sequence is determined by the state of the system.
- **Table Data Gateway**. An object that acts as a gateway to a table or view in a data source and centralizes all of the select, insert, update, and delete queries.
- **Table Module**. A single component that handles the business logic for all rows in a database table or view.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology and implement transaction support:

- If you require workflows that automatically support secure, reliable, transacted data exchange, a broad choice of transport and encoding options, and provide built-in persistence and activity tracking, consider using Windows Workflow (WF).

Additional Resources

- For more information, see *Concurrency Control* at http://msdn.microsoft.com/en-us/library/ms978457.aspx.
- For more information, see *Integration Patterns* at http://msdn.microsoft.com/en-us/library/ms978729.aspx.

Chapter 5 – Data Access Layer Guidelines

Objectives

- Understand how the data layer fits into the application architecture.
- Understand the components of the data layer.
- Learn the steps for designing these components.
- Learn the common issues faced when designing the data layer.
- Learn the key guidelines for designing the data layer.
- Learn the key patterns and technology considerations for designing the data access layer.

Overview

This chapter describes the key guidelines for designing the data layer of an application. The guidelines are organized by category and cover the common issues encountered, and mistakes commonly made, when designing the data layer. Figure 1. shows how the data layer fits into typical application architecture.

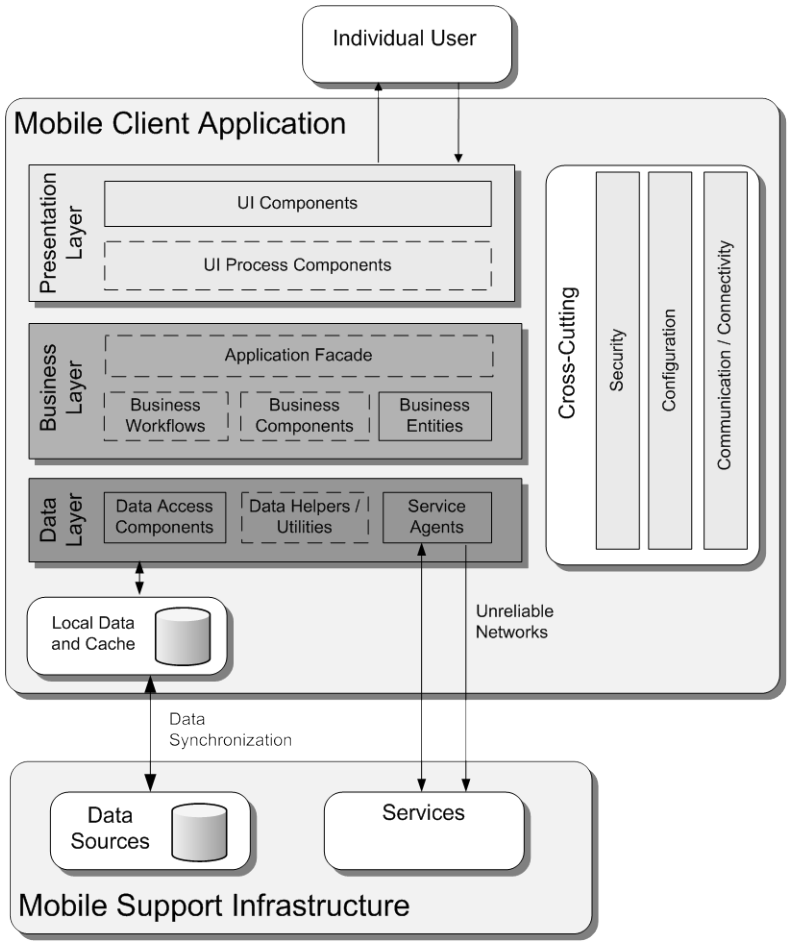

Figure 1 A typical application showing the data layer and the components it may contain

Data Layer Components

- **Data access logic components**. Data access components abstract the logic necessary to access your underlying data stores. Doing so centralizes the data access functionality, which makes the application easier to configure and maintain.
- **Data helpers / utilities**. Helper functions and utilities assist in data manipulation, data transformation, and data access within the layer. They consist of specialized libraries and/or custom routines especially designed to maximize data access performance and reduce the development requirements of the logic components and the service agent parts of the layer.
- **Service agents**. When a business component must use functionality exposed by an external service, you might need to create code that manages the semantics of communicating with that service. Service agents isolate your application from the idiosyncrasies of calling diverse services, and can provide additional services such as basic mapping between the format of the data exposed by the service and the format your application requires.

Approach

A correct approach to designing the data layer will reduce development time and assist in maintenance of the data layer after the application is deployed. This section briefly outlines an effective design approach for the data layer. Perform the following key activities in each of these areas when designing your data layer:

1. **Create an overall design for your data access layer:**
    a. Identify your data source requirements.
    b. Determine your data access approach.
    c. Choose how to map data structures to the data source.
    d. Determine how to connect to the data source.
    e. Determine strategies for handling data source errors.
2. **Design your data access components**:
    a. Enumerate the data sources that you will access.
    b. Decide on the method of access for each data source.
    c. Determine whether helper components are required or desirable to simplify data access component development and maintenance.
    d. Determine relevant design patterns. For example, consider using the Table Data Gateway, Query Object, Repository, and other patterns.
3. **Design your data helper components:**
    a. Identify functionality that could be moved out of the data access components and centralized for reuse.
    b. Research available helper component libraries.
    c. Consider custom helper components for common problems such as connection strings, data source authentication, monitoring, and exception processing.
    d. Consider implementing routines for data access monitoring and testing in your helper components.
    e. Consider the setup and implementation of logging for your helper components.

4. **Design your service agents:**
    a. Use the appropriate tool to add a service reference. This will generate a proxy and the data classes that represent the data contract from the service.
    b. Determine how the service will be used in your application. For most applications, you should use an abstraction layer between the business layer and the data access layer, which will provide a consistent interface regardless of the data source. For smaller applications, the business layer, or even the presentation layer, may access the service agent directly.

Design Guidelines

The following design guidelines provide information about different aspects of the data access layer that you should consider. Follow these guidelines to ensure that your data access layer meets the requirements of your application, performs efficiently and securely, and is easy to maintain and extend as business requirements change.

- **Choose the data access technology**. The choice of an appropriate data access technology will depend on the type of data you are dealing with, and how you want to manipulate the data within the application. Certain technologies are better suited for specific scenarios. Refer to the Data Access Technology Matrix found later in this guide. It discusses these options and enumerates the benefits and considerations for each data access technology.
- **Use abstraction to implement a loosely coupled interface to the data access layer**. This can be accomplished by defining interface components, such as a gateway with well-known inputs and outputs, which translate requests into a format understood by components within the layer. In addition, you can use interface types or abstract base classes to define a shared abstraction that must be implemented by interface components.
- **Consider consolidating data structures**. If you are dealing with table-based entities in your data access layer, consider using Data Transfer Objects (DTOs) to help you organize the data into unified structures. In addition, DTOs encourage coarse-grained operations while providing a structure that is designed to move data across different boundary layers.
- **Encapsulate data access functionality within the data access layer**. The data access layer hides the details of data source access. It is responsible for managing connections, generating queries, and mapping application entities to data source structures. Consumers of the data access layer interact through abstract interfaces using application entities such as custom objects, DataSets, DataReaders, and XML. Other application layers that access the data access layer will manipulate this data in more complex ways to implement the functionality of the application. Separating concerns in this way assists in application development and maintenance.
- **Decide how to map application entities to data source structures**. The type of entity you use in your application is the main factor in deciding how to map those entities to data source structures.
- **Decide how you will manage connections**. As a rule, the data access layer should create and manage all connections to all data sources required by the application. You must choose an appropriate method for storing and protecting connection information that conforms to application and security requirements.

- **Determine how you will handle data exceptions**. The data access layer should catch and (at least initially) handle all exceptions associated with data sources and CRUD (Create, Read, Update, and Delete) operations. Exceptions concerning the data itself, and data source access and timeout errors, should be handled in this layer and passed to other layers only if the failures affect application responsiveness or functionality.
- **Consider security risks**. The data access layer should protect against attacks that try to steal or corrupt data, and protect the mechanisms used to gain access to the data source. It should also use the "least privilege" design approach to restrict privileges to only those needed to perform the operations required by the application. If the data source itself has the ability to limit privileges, security should be considered and implemented in the data access layer as well as in the source.
- **Reduce round trips**. Consider batching commands into a single database operation.
- **Consider performance objectives**. Performance objectives for the data access layer should be taken into account during design.

Data Access Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Category	Common issues
BLOB	• Improperly storing BLOBs in the database instead of the file system. • Using an incorrect type for BLOB data in the database. • Searching and manipulating BLOB data.
Batching	• Failing to use batching to reduce database round trips. • Holding onto locks for excessive periods when batching.
Connections	• Failing to handle connection timeouts and disconnections. • Performing transactions that span multiple connections.
Data Format	• Choosing the wrong data format. • Failing to consider serialization requirements. • Not mapping objects to a relational data store.
Exception Management	• Not handling data access exceptions. • Failing to shield database exceptions from the original caller. • Failing to log critical exceptions.
Queries	• Using string concatenation to build queries. • Mixing queries with business logic. • Not optimizing the database for query execution.
Transactions	• Using the incorrect isolation level. • Using exclusive locks, which can cause contention and deadlocks. • Allowing long-running transactions to block access to data.

Validation	• Failing to validate and constrain data fields. • Not handling NULL values. • Not filtering for invalid characters.
XML	• Not considering how to handle extremely large XML data sets. • Not choosing the appropriate technology for XML to relational database interaction. •

BLOB

A BLOB is a binary large object. When data is stored and retrieved as a single stream of data, it can be considered to be a BLOB. A BLOB may have structure within it, but that structure is not apparent to the database that stores it or the data layer that reads and writes it. Databases can store the BLOB data or can store pointers to them within the database. The BLOB data is usually stored in a file system if not stored directly in the database. BLOBs are typically used to store image data, but can also be used to store binary representations of objects.

Consider the following guidelines when designing for BLOBs:

- Store BLOB data in a database only when it is not practical to store it on the file system.
- Consider using BLOBs to simplify synchronization of large binary objects between servers.
- Consider whether you need to search the BLOB data. If so, create and populate other searchable database fields instead of parsing the BLOB data.
- When retrieving the BLOB, cast it to the appropriate type for manipulation within your business or presentation layer.

Batching

Batching database commands can improve the performance of your data layer. Each request to the database execution environment incurs an overhead. Batching can reduce the total overhead by increasing throughput and decreasing latency. Batching similar queries can improve performance because the database caches and can reuse a query execution plan for a similar query.

Consider the following guidelines when designing batching:

- Consider using batched commands to reduce round trips to the database and minimize network traffic.
- Batch similar queries for maximum benefit. Batching dissimilar or random queries provides less reduction in overhead.
- Consider using batched commands and a DataReader to load or copy multiple sets of data.
- When loading large volumes of file-based data into the database, consider using bulk copy utilities.
- Do not consider placing locks on long-running batch commands.

Connections

Connections to data sources are a fundamental part of the data layer. All data source connections should be managed by the data layer. Creating and managing connections uses valuable resources in both the data layer and the data source. To maximize performance, follow guidelines for creating, managing, and closing connections

Consider the following guidelines when designing for data layer connections:

- In general, open connections as late as possible and close them as early as possible. If you are using SQL Server CE open the connection when application is launched and maintain it until the application is running.
- To maximize the effectiveness, consider using a trusted subsystem security model.
- Perform transactions through a single connection where possible.
- Design retry logic to manage the situation where the connection to the data source is lost or times out.

Data Format

Data formats and types are important in order to properly interpret the raw bytes stored in the database and transferred by the data layer. Choosing the appropriate data format provides interoperability with other applications, and facilitates serialized communications across different processes and physical machines. Data format and serialization are also important in order to allow the storage and retrieval of application state by the business layer.

Consider the following guidelines when designing your data format:

- In most cases, you should use custom data or business entities for improved application maintainability. This will require additional code to map the entities to database operations.
- Consider using XML for interoperability with other systems and platforms or when working with data structures that can change over time.
- Consider using DataSets for disconnected scenarios in simple CRUD-based applications.
- Understand the serialization and interoperability requirements of your application.

Exception Management

Design a centralized exception-management strategy so that exceptions are caught and thrown consistently in your data layer. If possible, centralize exception-handling logic in your database helper components. Pay particular attention to exceptions that propagate through trust boundaries and to other layers or tiers. Design for unhandled exceptions so they do not result in application reliability issues or exposure of sensitive application information.

Consider the following guidelines when designing your exception-management strategy:

- Determine exceptions that should be caught and handled in the data access layer. Deadlocks, connection issues, and optimistic concurrency checks can often be resolved at the data layer.

- Consider implementing a retry process for operations where data source errors or timeouts occur, where it is safe to do so.
- Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer.
- Design an approach for catching and handling unhandled exceptions.
- Design an appropriate logging and notification strategy for critical errors and exceptions that does not reveal sensitive information.

Queries

Queries are the primary data manipulation operations for the data layer. They are the mechanism that translates requests from the application into create, retrieve, update and delete (CRUD) actions on the database. As queries are so essential, they should be optimized to maximize database performance and throughput.

When using queries in your data layer, consider the following guidelines:

- Consider that SQL Server Compact Edition supports only a subset of T-SQL supported by SQL Server.
- Use parameterized SQL statements and typed parameters to mitigate security issues and reduce the chance of SQL injection attacks succeeding.
- When it is necessary to build queries dynamically, ensure that you validate user input data used in the query.
- Do not use string concatenation to build dynamic queries in the data layer.
- Consider using objects to build queries. For example, implement the Query Object pattern or use the object support provided by ADO.NET.
- When building dynamic SQL, avoid mixing business-processing logic with logic used to generate the SQL statement. Doing so can lead to code that is very difficult to maintain and debug.

Transactions

A *transaction* is an exchange of sequential information and associated actions that are treated as an atomic unit in order to satisfy a request and ensure database integrity. A transaction is only considered complete if all information and actions are complete, and the associated database changes are made permanent. Transactions support undo (rollback) database actions following an error, which helps to preserve the integrity of data in the database.

Consider the following guidelines when designing transactions:

- Enable transactions only when you need them. For example, you should not use a transaction for an individual SQL statement because Microsoft SQL Server® Compact Edition automatically executes each statement as an individual transaction.
- Keep transactions as short as possible to minimize the amount of time that locks are held.
- Use the appropriate isolation level. The tradeoff is data consistency versus contention. A high isolation level will offer higher data consistency at the price of overall concurrency. A

lower isolation level improves performance by lowering contention at the cost of consistency.
- Consider that Microsoft SQL Server Compact Edition does not support nested transactions.
- Do not use distributed transactions as it is not supported by Microsoft Compact Framework.

Validation

Designing an effective input and data-validation strategy is critical to the security of your application. Determine the validation rules for data received from other layers and from third-party components, as well as from the database or data store. Understand your trust boundaries so that you can validate any data that crosses these boundaries.

Consider the following guidelines when designing a validation strategy:
- Validate all data received by the data layer from all callers.
- Consider the purpose to which data will be put when designing validation. For example, user input used in the creation of dynamic SQL should be examined for characters or patterns that occur in SQL injection attacks.
- Understand your trust boundaries so that you can validate data that crosses these boundaries.
- Return informative error messages if validation fails.

XML

Extensible Markup Language (XML) is useful for interoperability and for maintaining data structure outside the database. For performance reasons, be careful when using XML for very large amounts of data. If you must handle large amounts of data, use attribute-based schemas instead of element-based schemas. Use schemas to validate the XML structure and content.

Consider the following guidelines when designing for the use of XML:
- Consider using XML readers and writers to access XML-formatted data.
- Consider using custom validators for complex data parameters within your XML schema.
- Store XML in typed columns in the database, if available, for maximum performance.

Manageability Considerations

Manageability is an important factor in your application because a manageable application is easier for administrators and operators to install, configure, and monitor. Manageability also makes it easier to detect, validate, resolve, and verify errors at run time. You should always strive to maximize manageability when designing your application.

Consider the following guidelines when designing for manageability:
- Consider using common interface types or a shared abstraction (Dependency Inversion) to provide an interface to the data access layer.
- Consider the use of custom entities, or decide if other data representations will better meet your requirements. Coding custom entities can increase development costs; however, they

also provide improved performance through binary serialization and a smaller data footprint.
- Implement business entities by deriving them from a base class that provides basic functionality and encapsulates common tasks. However, be careful not to overload the base class with unrelated operations, which would reduce the cohesiveness of entities derived from the base class, and cause maintainability and performance issues.
- Design business entities to rely on data access logic components for database interaction. Centralize implementation of all data access policies and related business logic. For example, if your business entities access the databases directly, all applications deployed to clients that use the business entities will require SQL connectivity and logon permissions.

Performance Considerations

Performance is a function of both your data layer design and your database design. Consider both together when tuning your system for maximum data throughput.

Consider the following guidelines when designing for performance:
- Tune performance based on results obtained by running simulated load scenarios.
- Consider tuning isolation levels for data queries. If you are building an application with high-throughput requirements, special data operations may be performed at lower isolation levels than the rest of the transaction. Combining isolation levels can have a negative impact on data consistency, so you must carefully analyze this option on a case-by-case basis.
- Consider batching commands to reduce round trips to the database server.
- Consider using optimistic concurrency with non-volatile data to mitigate the cost of locking data in the database. This avoids the overhead of locking database rows, including the connection that must be kept open during a lock.
- If using a DataReader, use ordinal lookups for faster performance.

Security Considerations

The data layer should protect the database against attacks that try to steal or corrupt data. It should allow only as much access to the various parts of the data source as is required. The data layer should also protect the mechanisms used to gain access to the data source.

Consider the following guidelines when designing for security:
- Consider using Windows authentication where possible.
- Encrypt connection strings in configuration files instead of using a system name.
- When storing passwords, use a salted hash instead of an encrypted version of the password.
- Require that callers send identity information to the data layer for auditing purposes.
- If you are using SQL statements, consider the parameterized approach instead of string concatenation to protect against SQL injection attacks.

Pattern Map

Key patterns are organized by the key categories detailed in the Data Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Category	Relevant patterns
General	• Active Record • Data Transfer Object • Query Object • Table Data Gateway • Table Module
Batching	• Parallel Processing • Partitioning
Transactions	• Coarse-Grained Lock • Implicit Lock • Optimistic Offline Lock • Pessimistic Offline Lock • Transaction Script

- For more information on the Table Module, Coarse-Grained Lock, Implicit Lock, Transaction Script, Active Record, Data Transfer Object, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository patterns, see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/

Pattern Descriptions

- **Active Record**. Include a data access object within a domain entity.
- **Coarse Grained Lock**. Lock a set of related objects with a single lock.
- **Data Mapper.** Implement a mapping layer between objects and the database structure that is used to move data from one structure to another while keeping them independent.
- **Data Transfer Object**. An object that stores the data transported between processes, reducing the number of method calls required.
- **Implicit Lock**. Use framework code to acquire locks on behalf of code that accesses shared resources.
- **Optimistic Offline Lock**. Ensure that changes made by one session do not conflict with changes made by another session.
- **Parallel Processing**. Allow multiple batch jobs to run in parallel to minimize the total processing time.
- **Partitioning**. Partition multiple large batch jobs to run concurrently.
- **Pessimistic Offline Lock**. Prevent conflicts by forcing a transaction to obtain a lock on data before using it.
- **Query Object**. An object that represents a database query.
- **Table Data Gateway**. An object that acts as a gateway to a table in a data source and centralizes all of the select, insert, update, and delete queries.

- **Table Module**. A single component that handles the business logic for all rows in a database table.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology and techniques depending on the type of application you are designing and the requirements of that application:

- If you require basic support for queries and parameters, consider using ADO.NET objects directly.
- If you want to manipulate XML-formatted data, consider using the classes in the **System.Xml** namespace and its subsidiary namespaces.
- If you are accessing Microsoft SQL Server Compact Edition, consider using classes in the ADO.NET SqlClient namespace to maximize performance.
- If you are designing an object-oriented business layer based on the Domain Model pattern, consider using the ADO.NET Entity Framework.

Additional Resources

For more information on general data access guidelines, see the following resources:

- *Typing, storage, reading, and writing BLOBs* at http://msdn.microsoft.com/en-us/library/ms978510.aspx#daag_handlingblobs
- *.NET Data Access Architecture Guide* at http://msdn.microsoft.com/en-us/library/ms978510.aspx.
- *Data Patterns* at http://msdn.microsoft.com/en-us/library/ms998446.aspx.
- *Designing Data Tier Components and Passing Data Through Tiers* at http://msdn.microsoft.com/en-us/library/ms978496.aspx

Chapter 6: Service Layer Guidelines

Objectives

- Understand how the service layer fits into your mobile application architecture.
- Understand the components of the service layer.
- Learn the steps for designing the service layer.
- Learn the common issues faced when designing the service layer.
- Learn the key guidelines for designing the service layer.
- Learn the key patterns and technology considerations for designing the service layer.

Overview

A mobile device normally includes a remote server infrastructure to support business functions and manage mobile application updates. Services are often used to communicate with the remote servers. When providing application functionality through services, it is important to separate the service functionality into a separate service layer. Within the service layer, you define the service interface, implement the service interface, and provide translator components that translate data formats between the business layer of your server infrastructure and external data contracts. One of the more important concepts to keep in mind is that a service should never expose internal entities that are used by the business layer. Figure 1 shows where a service layer fits into the overall design of your application. In this figure below, a mobile device would be considered an “External System”.

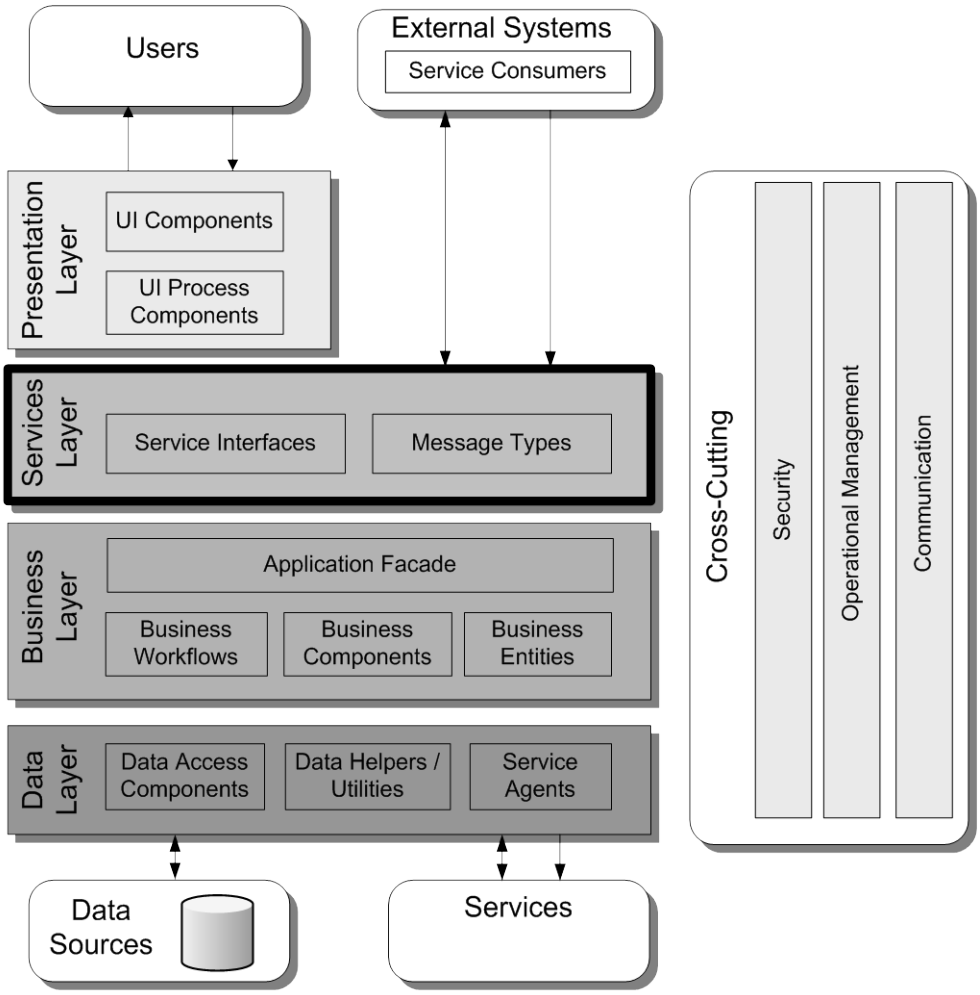

Figure 1 An overall view of a typical mobile application infrastructure showing the service layer

Service Layer Components

The service layer is made up of the following components:

- **Service interfaces**. Services expose a service interface to which all inbound messages are sent. The definition of the set of messages that must be exchanged with a service, in order for the service to perform a specific business task, constitutes a contract. You can think of a service interface as a façade that exposes the business logic implemented in the service to potential consumers.
- **Message types**. When exchanging data across the service layer, data structures are wrapped by message structures that support different types of operations. For example, you might have a Command message, a Document message, or another type of message. These message types are the "message contracts" for communication between service consumers and providers.

Approach

The approach used to design a service layer starts by defining the service interface, which consists of the contracts that you plan to expose from your service. Once the service interface

has been defined, the next step is to design the service implementation; which is used to translate data contracts into business entities and to interact with the business layer.

The following steps can be used when designing a service layer:

- Define the Data and Message contracts that represent the schema used for messages.
- Define the Service contracts that represent operations supported by your service.
- Define the Fault contracts that return error information to consumers of the service.
- Design transformation objects that translate between business entities and data contracts.
- Design the abstraction approach used to interact with the business layer.

Design Considerations

There are many factors that you should consider when designing the service layer. Many of these design considerations relate to proven practices concerned with layered architectures. However, with a service, you must take into account message-related factors. The main thing to consider is that a service uses message-based interaction, which is inherently slower than object-based interaction. In addition, messages passed between a service and a consumer can be routed, modified, or lost, which requires a design that will account for the non-deterministic behavior of messaging. Often, your service layer will support both mobile and non-mobile clients. It may be important to provide mobile-only interfaces in some cases. Mobile clients will often support only a subset of the communication mechanisms and will have limited local storage ability compared to desktop and laptop clients.

Consider the following guidelines when designing the service layer:

- **Design services to be application-scoped and not component-scoped**. Service operations should be coarse-grained and focused on application operations. For example, with demographics data, you should provide an operation that returns all of the data in one call. You should not use multiple operations to return subsets of the data with multiple calls.
- **Design entities for extensibility**. In other words, data contracts should be designed so that you can extend them without affecting consumers of the service.
- **Compose entities from standard elements**. When possible, use standard elements to compose the complex types used by your service.
- **Use a layered approach to designing services**. Separate the business rules and data access functions into distinct components where appropriate.
- **Avoid tight coupling across layers**. Use abstraction to provide an interface into the business layer. This abstraction can be implemented by using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation and business layers.
- **Design only for the service contract**. In other words, you should not implement functionality that is not reflected by the service contract. In addition, the implementation of a service should never be exposed to external consumers.
- **Design to assume the possibility of invalid requests**. You should never assume that all messages received by the service will be valid.

- **Separate functional business concerns from infrastructure operational concerns**. Cross-cutting logic should never be combined with application logic. Doing so can lead to implementations that are difficult to extend and maintain.
- **Ensure that the service can detect and manage repeated messages (idempotency)**. When designing the service, implement well-known patterns to ensure that duplicate messages are not processed.
- **Ensure that the service can manage messages arriving out of order (commutativity)**. If there is a possibility that messages will arrive out of order, implement a design that will store messages and then process them in the correct order.
- **Consider versioning of contracts**. A new version for service contracts mean new operations exposed by the service, whereas for data contracts it means the addition of new schema type definitions.
- **Design for occasional connected devices**. As a device is more likely to lose connectivity, design your services to deal with this possibility. Do not rely on long running communications.
- **Consider the size of data being transferred.** Devices have limited memory and cannot store very large amounts of data. Consider minimizing the raw number of bytes being transferred to a device.

Service Layer Frame

There are several common issues that you must consider as your develop your service layer design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Area	Key issues
Authentication and Authorization	• Lack of authentication across trust boundaries. • Lack of authorization across trust boundaries. • Granular or improper authorization.
Communication	• Incorrect choice of transport protocol. • Use of a chatty service communication interface. • Failing to protect sensitive data.
Exception Management	• Not catching exceptions that can be handled. • Not logging exceptions. • Not dealing with message integrity when an exception occurs.
Messaging Channels	• Choosing an inappropriate message channel • Failing to handle exception conditions on the channel. • Providing access to non-messaging clients.
Message Construction	• Failing to handle time-sensitive message content. • Incorrect message construction for the operation. • Passing too much data in a single message.
Message Endpoint	• Not supporting idempotent operations. • Not supporting commutative operations. • Subscribing to an endpoint while disconnected.

Area	Key issues
Message Protection	• Not protecting sensitive data. • Not using transport layer protection for messages that cross multiple servers. • Not considering data integrity.
Message Routing	• Not choosing the appropriate router design. • Ability to access a specific item from a message. • Ensuring that messages are handled in the correct order.
Message Transformation	• Performing unnecessary transformations. • Implementing transformations at the wrong point. • Using a canonical model when not necessary.
REST	• Implementing state within the service. • Overusing POST statements. • Putting actions in the URI. • Using hypertext to manage state.
SOAP	• Not choosing the appropriate security model. • Not planning for fault conditions. • Using complex types in the message schema.

Authentication

Designing an effective authentication strategy for your service layer is important for the security and reliability of your application. Failure to design a good authentication strategy can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attacks.

Consider the following guidelines when designing an authentication strategy:

- Identify a suitable mechanism for securely authenticating users.
- Consider the implications of using different trust settings for executing service code.
- Ensure that secure protocols such as Secure Sockets Layer (SSL) are used with Basic authentication, or when credentials are passed as plain text.
- Consider using secure mechanisms such as WS Security with SOAP messages.

Authorization

Designing an effective authorization strategy for your service layer is important for the security and reliability of your application. Failure to design a good authorization strategy can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges.

Consider the following guidelines when designing an authorization strategy:

- Set appropriate access permissions on resources for users, groups, and roles.
- Use URL authorization and/or file authorization when using Windows authentication.
- Where appropriate, restrict access to publicly accessible Web methods by using declarative principle permission demands.
- Execute services under the most restrictive account that is appropriate.

Communication

When designing the communication strategy for your service, the protocol you choose should be based on the deployment scenario your service must support. If the service will be deployed within a closed network, you can use Transmission Control Protocol (TCP) for more efficient communications. If the service will be deployed into a public-facing network, consider using HyperText Transfer Protocol (HTTP) protocol. Consider the different devices and how to support differing communication capabilities of those devices that may be connecting to your services.

Consider the following guidelines when designing a communication strategy:

- Determine how to handle unreliable or intermittent communication.
- Minimize the number of bytes transferred to make communication with mobile devices efficient and less likely to be interrupted. Use efficient transfer mechanisms and avoid long-running communications when a connection could be unreliable.
- Consider using dynamic URL behavior with configured endpoints for maximum flexibility.
- Validate endpoint addresses in messages.
- Determine whether you need to make asynchronous calls.
- Determine if you need request-response or duplex communication. However, keep in mind that duplex communications require a consistent reliable connection.

Exception Management

Designing an effective exception-management strategy for your service layer is important for the security and reliability of your application. Failure to do so can make your application vulnerable to denial of service (DoS) attacks, and can also allow it to reveal sensitive and critical information.

Raising and handling exceptions is an expensive operation, so it is important for the design to take into account the potential impact on performance. A good approach is to design a centralized exception management and logging mechanism, and consider providing access points that support instrumentation and centralized monitoring in order to assist system administrators.

Consider the following guidelines when designing an exception-management strategy:

- Do not use exceptions to control business logic.
- Design a strategy for handling unhandled exceptions.
- Do not reveal sensitive information in exception messages or log files.
- Use SOAP Fault elements or custom extensions to return exception details to the caller.
- Disable tracing and debug-mode compilation for all services, except during development and testing.

Messaging Channels

Communication between a service and its consumers consists of sending data through a channel. In most cases, you will use channels provided by your chosen service infrastructure, such as Windows Communication Foundation (WCF). You must understand which patterns your

chosen infrastructure supports, and determine the appropriate channel for interaction with consumers of the service.

Consider the following guidelines when designing message channels:

- Determine appropriate patterns for messaging channels, such as Channel Adapter, Messaging Bus, and Messaging Bridge.
- Determine how you will intercept and inspect the data between endpoints if necessary.
- Verify that you mobile device supports the same channel formats as your service. For example, Compact WCF supports only a subset of all the possible Desktop WCF features. Verify supported Bindings, Formatters, Encoders, Transports, ws-* standards, etc.

Message Construction

When data is exchanged between a service and consumer, it must be wrapped inside a message. The format of that message is based on the types of operations you need to support. For example, you may be exchanging documents, executing commands, or raising events. When using slow message-delivery channels, you should also consider using expiration information in the message.

Consider the following guidelines when designing a message-construction strategy:

- Determine the appropriate patterns for message constructions, such as Command, Document, Event, and Request-Reply.
- Divide very large quantities of data into smaller chunks, and send them in sequence.
- Include expiration information in messages that are time-sensitive. The service should ignore expired messages.

Message Endpoint

The message endpoint represents the connection that applications use to interact with your service. The implementation of your service interface represents the message endpoint. When designing the service implementation, you must consider the possibility that duplicate or invalid messages can be sent to your service.

Consider the following guidelines when designing message endpoints:

- Determine relevant patterns for message endpoints, such as Gateway, Mapper, Competing Consumers, and Message Dispatcher.
- Determine if you should accept all messages or implement a filter to handle specific messages.
- Design for idempotency in your message interface. Idempotency is the situation where you could receive duplicate messages from the same consumer, but should only handle one. In other words, an idempotent endpoint will guarantee that only one message will be handled, and all duplicate messages will be ignored.
- Design for commutativity in your message interface. Commutativity is related to the order in which messages are received. In some cases, you may need to store inbound messages so that they can be processed in the correct order.

- Design for disconnected scenarios. For instance, you might need to support guaranteed delivery.

Message Protection

When transmitting sensitive data between a service and its consumer, you should design for message protection. You can use transport layer protection or message-based protection. However, in most cases, you should use message-based protection. For example, you should encrypt sensitive sections within a message and use a signature to protect the message from tampering.

Consider the following guidelines when designing message protection:

- Be sure that your services use only mobile supported protection mechanisms
- If interactions between the service and the consumer are not routed through other services, you can use transport layer security only, such as SSL.
- If the message passes through one or more servers, always use message-based protection. In addition, you can also use transport layer security with message-based security. With transport layer security, the message is decrypted and then encrypted at each server it passes through, which represents a security risk.
- Consider using both transport layer and message-based security in your design.
- Use encryption to protect sensitive data in messages.
- Consider using digital signatures to protect messages and parameters from tampering.

Message Routing

A message router is used to decouple a service consumer from the service implementation. There are three main types of routers that you might use: simple, composed, and pattern-based. Simple routers use a single router to determine the final destination of a message. Composed routers combine multiple simple routers to handle more complex message flows. Architectural patterns are used to describe different routing styles based on simple message routers.

Consider the following guidelines when designing message routing:

- Determine relevant patterns for message routing, such as Aggregator, Content-Based Router, Dynamic Router, and Message Filter.
- If sequential messages are sent from a consumer, such as a mobile device, the router must ensure that they are all delivered to the same endpoint in the required order (commutativity).
- A message router will normally inspect information in the message to determine how to route the message. As a result, you must ensure that the router can access that information.

Message Transformation

When passing messages between a service and consumer, there are many cases where the message must be transformed into a format that the consumer can understand. This normally occurs in cases where non–message-based consumers need to process data from a message-based system. You can use adapters to provide access to the message channel for a non–message-based consumer, and translators to convert the message data into a format that the consumer understands.

Consider the following guidelines when designing message transformation:

- Determine relevant patterns for message transformation, such as Canonical Data Mapper, Envelope Wrapper, and Normalizer.
- Use metadata to define the message format.
- Consider using an external repository to store the metadata.

REST

Representational State Transfer (REST) represents an architecture style for distributed systems. It is designed to reduce complexity by dividing a system into resources. The resources and the operations supported by a resource are represented and exposed as a set of URIs over the HTTP protocol.

Consider the following guidelines when designing REST resources:

- Identify and categorize resources that will be available to clients.
- Choose an approach for resource representation. A good practice would be to use meaningful names for REST starting points and unique identifiers, such as a globally unique identifier (GUID), for specific resource instances. For example, http://www.contoso.com/employee/ represents an employee starting point. http://www.contoso.com/employee/8ce762d5-b421-6123-a041-5fbd07321bac4 uses a GUID that represents a specific employee.
- Decide if multiple representations should be supported for different resources. For example, you can decide if the resource should support an XML, Atom, or JSON format and make it part of the resource request. A resource could be exposed as both http://www.contoso.com/example.atom and http://www.contoso.com/example.json
- Decide if multiple views should be supported for different resources. For example, decide if the resource should support GET and POST operations, or only GET operations.
- Protect against scenarios where your REST could potentially send large amounts of data directly to a mobile device. Either build limitations into the service as a whole or provide a set of mobile specific URLs where some implement protected behavior. For example, if http://www.contoso.com/employee/ returns a set of all possible employees and some limited details, you could easily overload a mobile device returning tens of thousands of records in an XML format.

Service Interface

The service interface represents the contract exposed by your service. When designing a service interface, you should consider boundaries that must be crossed and the type of consumers who will be accessing your service. For instance, service operations should be coarse-grained and application scoped. One of the biggest mistakes with service interface design is to treat the service as a component with fine-grained operations. This results in a design that requires multiple calls across physical or process boundaries, which are very expensive in terms of performance and latency.

Consider the following guidelines when designing a service interface:

- Consider using a coarse-grained interface to batch requests and minimize the number of calls over the network.
- Design service interfaces in such a way that changes to the business logic do not affect the interface.
- Do not implement business rules in a service interface.
- Consider using standard formats for parameters to provide maximum compatibility with different types of clients.
- Do not make assumptions in your interface design about the way that clients will use the service.
- Do not use object inheritance to implement versioning for the service interface.

SOAP

SOAP is a message-based protocol that is used to implement the message layer of a service. The message is composed of an envelope that contains a header and body. The header can be used to provide information that is external to the operation being performed by the service. For instance, a header may contain security, transaction, or routing information. The body contains contracts, in the form of XML schemas, which are used to implement the service.

Consider the following guidelines when designing SOAP messages:

- Define the schema for the operations that can be performed by a service.
- Define the schema for the data structures passed with a service request.
- Define the schema for the errors or faults that can be returned from a service request.

Deployment Considerations

The service layer can be deployed on the same tier as other layers of the application, or on a separate tier in cases where performance and isolation requirements demand this. However, in most cases the service layer will reside on the same physical tier as the business layer in order to minimize performance impact when exposing business functionality.

Consider the following guidelines when deploying the service layer:

- Deploy the service layer to the same tier as the business layer to improve application performance, unless performance and security issues inherent within the production environment prevent this.
- If the service is accessed only by other applications within a local network, consider using TCP for communications, assuming your mobile devices are also able to support TCP communications
- If the service is publicly accessible from the Internet, use HTTP for your transport protocol.

Pattern Map

Key patterns are organized by the key categories detailed in the Service Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Category	Relevant patterns
Communication	• Duplex • Fire and Forget • Reliable Sessions • Request Response
Messaging Channels	• Channel Adapter • Message Bus • Messaging Bridge • Point-to-point Channel • Publish-subscribe Channel
Message Construction	• Command Message • Document Message • Event Message • Request-Reply
Message Endpoint	• Competing Consumer • Durable Subscriber • Idempotent Receiver • Message Dispatcher • Messaging Gateway • Messaging Mapper • Polling Consumer • Selective Consumer • Service Activator • Transactional Client
Message Protection	• Data Confidentiality • Data Integrity • Data Origin Authentication • Exception Shielding • Federation • Replay Protection • Validation

Category	Relevant patterns
Message Routing	• Aggregator • Content-Based Router • Dynamic Router • Message Broker (Hub-and-Spoke) • Message Filter • Process Manager
Message Transformation	• Canonical Data Mapper • Claim Check • Content Enricher • Content Filter • Envelope Wrapper • Normalizer
REST	• Behavior • Container • Entity • Store • Transaction
Service Interface	• Remote Façade
SOAP	• Data Contracts • Fault Contracts • Service Contracts

- For more information on the Duplex and Request Reponse patterns, see “Designing Service Contracts” at http://msdn.microsoft.com/en-us/library/ms733070.aspx
- For more information on the Atomic and Cross-Service Transaction patterns, see “WS-* Specifications” at http://www.ws-standards.com/ws-atomictransaction.asp
- For more information on the Command, Document Message, Event Message, Durable Subscriber, Idempotent Receiver, Polling Consumer, and Transactional Client patterns, see “Messaging Patterns in Service-Oriented Architecture, Part I” at http://msdn.microsoft.com/en-us/library/aa480027.aspx
- For more information on the Data Confidentiality and Data Origin Authentication patterns, see “Chapter 2: Message Protection Patterns” at http://msdn.microsoft.com/en-us/library/aa480573.aspx
- For more information on the Replay Detection, Exception Shielding, and Validation patterns, see “Chapter 5: Service Boundary Protection Patterns” at http://msdn.microsoft.com/en-us/library/aa480597.aspx
- For more information on the Claim Check, Content Enricher, Content Filter, and Envelope Wrapper patterns, see ”Messaging Patterns in Service Oriented Architecture, Part 2” at http://msdn.microsoft.com/en-us/library/aa480061.aspx
- For more information on the Remote Façade pattern, see “P of EAA: Remote Façade” at http://martinfowler.com/eaaCatalog/remoteFacade.html
- For more information on *REST* patterns such as Behavior, Container, and Entity, see “REST Patterns” at http://wiki.developer.mindtouch.com/REST/REST_Patterns

For more information on the Aggregator, Content-Based Router, Publish-Subscribe, Message Bus, and Point-to-Point patterns, see "Messaging patterns in Service-Oriented Architecture, Part I" at http://msdn.microsoft.com/en-us/library/aa480027.aspx

Pattern Descriptions

- **Aggregator**. A filter that collects and stores individual related messages, combines these messages, and publishes a single aggregated message to the output channel for further processing.
- **Behavior (REST)**. Applies to resources that carry out operations. These resources generally contain no state of their own, and only support the POST operation.
- **Canonical Data Mapper**. Uses a common data format to perform translations between two disparate data formats.
- **Channel Adapter**. A component that can access the application's API or data and publish messages on a channel based on this data, and that can receive messages and invoke functionality inside the application.
- **Claim Check**. Retrieves data from a persistent store when required.
- **Command Message**. Provides a message structure used to support commands.
- **Competing Consumer**. Sets multiple consumers on a single message queue and have them compete for the right to process the messages, which allows the messaging client to process multiple messages concurrently.
- **Container**. Builds on the entity pattern by providing the means to dynamically add and/or update nested resources.
- **Content Enricher**. Enriches messages with missing information obtained from an external data source.
- **Content Filter**. Removes sensitive data from a message and reduces network traffic by removing unnecessary data from a message.
- **Content-Based Router**. Routes each message to the correct consumer based on the contents of the message; such as existence of fields, specified field values, and so on.
- **Data Confidentiality**. Uses message-based encryption to protect sensitive data in a message.
- **Data Contract**. A schema that defines data structures passed with a service request.
- **Data Integrity**. Ensures that messages have not been tampered with in transit.
- **Data Origin Authentication**. Validates the origin of a message as an advanced form of data integrity.
- **Document Message**. A structure used to reliably transfer documents or a data structure between applications.
- **Duplex**. Two-way message communication where both the service and the client send messages to each other independently, irrespective of the use of the one-way or the request/reply pattern.
- **Durable Subscriber**. In a disconnected scenario, messages are saved and then made accessible to the client when connecting to the message channel in order to provide guaranteed delivery.

- **Dynamic Router**. A component that dynamically routes the message to a consumer after evaluating the conditions/rules that the consumer has specified.
- **Entity**. **(REST)**. Resources that can be read with a GET operation, but can only be changed by PUT and DELETE operations.
- **Envelope Wrapper**. A wrapper for messages that contains header information used, for example, to protect, route, or authenticate a message.
- **Event Message**. A structure that provides reliable asynchronous event notification between applications.
- **Exception Shielding**. Prevents a service from exposing information about its internal implementation when an exception occurs.
- **Façade**. Implements a unified interface to a set of operations in order to provide a simplified reduce coupling between systems.
- **Fault Contracts**. A schema that defines errors or faults that can be returned from a service request.
- **Federation**. An integrated view of information distributed across multiple services and consumers.
- **Fire and Forget**. A one-way message communication mechanism used when no response is expected.
- **Idempotent Receiver**. Ensures that a service will only handle a message once.
- **Message Broker (Hub-and-Spoke)**. A central component that communicates with multiple applications to receive messages from multiple sources, determines the correct destination, and route the message to the correct channel.
- **Message Bus**. Structures the connecting middleware between applications as a communication bus that enables the applications to work together using messaging.
- **Message Dispatcher**. A component that sends messages to multiple consumers.
- **Message Filter**. Eliminates undesired messages, based on a set of criteria, from being transmitted over a channel to a consumer.
- **Messaging Bridge**. A component that connects messaging systems and replicates messages between these systems.
- **Messaging Gateway**. Encapsulates message-based calls into a single interface in order to separate it from the rest of the application code.
- **Messaging Mapper**. Transforms requests into business objects for incoming messages, and reverses the process to convert business objects into response messages.
- **Normalizer**. Converts or transforms data into a common interchange format when organizations use different formats.
- **Point-to-point Channel**. Sends a message on a Point-to-Point Channel to ensure that only one receiver will receive a particular message.
- **Polling Consumer**. A service consumer that checks the channel for messages at regular intervals.
- **Process Manager**. A component that enables routing of messages through multiple steps in a workflow.
- **Publish-subscribe Channel**. Creates a mechanism to send messages only to the applications that are interested in receiving the messages without knowing the identity of the receivers.

- **Reliable Sessions**. End-to-end reliable transfer of messages between a source and a destination, regardless of the number or type of intermediaries that separate endpoints.
- **Remote Façade**. Creates a high-level unified interface to a set of operations or processes in a remote subsystem to make that subsystem easier to use, by providing a course-grained interface over fine-grained operations to minimize calls across the network.
- **Replay Protection**. Enforces message idempotency by preventing an attacker from intercepting a message and executing it multiple times.
- **Request Response**. A two-way message communication mechanism where the client expects to receive a response for every message sent.
- **Request-Reply**. Uses separate channels to send the request and reply.
- **Selective Consumer**. The service consumer uses filters to receive messages that match specific criteria.
- **Service Activator**. A service that receives asynchronous requests to invoke operations in business components.
- **Service Contract**. A schema that defines operations that the service can perform.
- **Service Interface**. A programmatic interface that other systems can use to interact with the service.
- **Store (REST)**. Allows entries to be created and updated with PUT.
- **Transaction (REST)**. Resources that support transactional operations.
- **Transactional Client**. A client that can implement transactions when interacting with a service.
- **Validation**. Checks the content and values in messages to protect a service from malformed or malicious content.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology for your service layer:

- Consider using ASP.NET Web services (ASMX) for simplicity, but only when a suitable Web server will be available.
- Consider using WCF services for advanced features and support for multiple transport protocols.
- If you are using ASMX and you require message-based security and binary data transfer, consider using Web Service Extensions (WSE).
- If you are using WCF and you want interoperability with non-WCF or non-Windows clients, consider using HTTP transport based on SOAP specifications.
- If you are using WCF, consider defining service contracts that use an explicit message wrapper instead of an implicit one. This allows you to define message contracts as inputs and outputs for your operations, which then allows you to extend the data contracts included in the message contract without affecting the service contract.

Additional Resources

- For more information, see *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn.microsoft.com/en-us/library/ms998469.aspx.
- For more information, see *Web Service Security Guidance* at http://msdn.microsoft.com/en-us/library/aa480545.aspx
- For more information, see *Improving Web Services Security: Scenarios and Implementation Guidance for WCF* at http://www.codeplex.com/WCFSecurityGuide

Chapter 7 - Communication Guidelines

Objectives

- Learn the guidelines for designing a communication approach for both the infrastructure supporting mobile devices and for the devices themselves.
- Learn the ways in which components communicate with each other.
- Learn the interoperability, performance, and security considerations for choosing a communication approach.
- Learn the communication technology choices.

Overview

One of the key factors that affect the design of an application, particularly a distributed application, is the way that you design the communication infrastructure for each part of the application. Mobile infrastructure support components must communicate with each other, for example to send user input to the business layer, and then to update the data store through the data layer. The mobile infrastructure must then communicate with the mobile device. When components are located on the same physical tier, you can often rely on direct communication between these components. However, if you deploy components and layers on physically separate servers and client machines - as is likely in most scenarios - you must consider how the components in these layers will communicate with each other efficiently and reliably. This will also be true on the mobile device, though the methods of communication may differ due to the smaller hardware footprint of the device. Always check the latest documentation for your version of Windows Mobile to see what subset of communication mechanisms is available on your device(s).

In general, you must choose between direct communication where components calls methods on each other, and message-based communication. There are many advantages to using message-based communication, such as decoupling and the capability to change your deployment strategy in the future. However, message-based communication raises issues that you must consider, such as performance, reliability, and - in particular - security.

This chapter contains design guidelines that will help you to choose the appropriate communication approach, understand how to get the most from it, and understand security and reliability issues that may arise.

Design Guidelines

When designing a communication strategy for your application, consider the performance impact of communicating between layers, as well as between tiers. Because each communication across a logical or a physical boundary increases processing overhead, design

for efficient communication by reducing round trips and minimizing the amount of data sent over the network.

- **Consider communication strategies when crossing boundaries.** Understand each of your boundaries, and how they affect communication performance. For example, the application domain (AppDomain), computer process, machine, and unmanaged code all represent boundaries that that can be crossed when communicating with components of the application or external services and applications.
- **Consider using unmanaged code for communication across AppDomain boundaries**. Use unmanaged code to communicate across AppDomain boundaries. This approach requires assemblies to run in full trust in order to interact with unmanaged code.
- **Consider using message-based communication when crossing process boundaries.** Use Windows Communication Foundation (WCF) with either the TCP to package data into a single call that can be serialized across process boundaries. On the mobile device itself, inter-process communication can use sockets, point to point queues, Windows Messages or MSMQ depending on the scenario.
- **Consider message-based communication when crossing physical boundaries.** In the infrastructure supporting mobile devices, consider using Windows Communication Foundation (WCF) or Microsoft Message Queuing (MSMQ) to communicate with remote machines across physical boundaries. Message-based communication supports coarse-grained operations that reduce round trips when communicating across a network. Between the mobile device and the infrastructure, consider using technologies that allow for the occasionally connected nature of mobile communications such as WCF Store and Forward.
- **Reduce round trips when accessing remote layers**. When communicating with remote layers, reduce communication requirements by using coarse-grained message-based communication methods, and use asynchronous communication if possible to avoid blocking or freezing the user interface.
- **Consider the serialization capabilities of the data formats passed across boundaries**. If you require interoperability with other systems, consider XML serialization. Keep in mind that XML serialization imposes increased overhead. If performance is critical, consider binary serialization because it is faster and the resulting serialized data is smaller than the XML equivalent.
- **Consider hotspots while designing your communication policy.** Hotspots include asynchronous and synchronous communication, data format, communication protocol, security, performance, and interoperability.

Message-Based Communication

Message-based communication allow you to expose a service to your callers by defining a service interface that clients call by passing XML-based messages over a transport channel. Message-based calls are generally made from remote clients, but message-based service interfaces can support local callers as well. A message-based communication style is well suited to the following scenarios:

- You are implementing a business system that represents a medium- to long-term investment; for example, when building a service that will be exposed to and used by partners for a considerable time.
- You are implementing large-scale systems with high availability characteristics.
- You are building a service that you want to isolate from other services it uses, and from services that consume it.
- You expect communication at either of the endpoints to be sporadically unavailable, as in the case of wireless networks or applications that can be used offline.
- You are dealing with real-world business processes that use the asynchronous model. This will provide a cleaner mapping between your requirements and the behavior of the application.

When using message-based communication, consider the following guidelines:

- Consider that a connection will not always be present, and messages may need to be stored and then sent when a connection becomes available.
- Consider how to handle the case when a message response is not received. To manage the conversation state, your business logic can log the sent messages for later processing in case a response is not received.
- Use acknowledgements to force the correct sequencing of messages.
- If message response timing is critical for your communication, consider a synchronous programming model in which your client waits for each response message.
- Do not implement a custom communication channel unless there is no default combination of endpoint, protocol, and format that suits your needs.

Asynchronous and Synchronous Communication

Consider the key tradeoffs when choosing between synchronous and asynchronous communication styles. Synchronous communication is best suited to scenarios in which you must guarantee the order in which calls are received, or when you must wait for the call to return before proceeding. Asynchronous communication is best suited to scenarios in which responsiveness is important or you cannot guarantee the target will be available.

Consider the following guidelines when deciding whether to use synchronous or asynchronous communication:

- For maximum performance, loose-coupling, and minimized system overhead, consider using an asynchronous communication model.
- Where you must guarantee the order in which operations take place, or you use operations that depend on the outcome of previous operations, consider a synchronous model.
- For asynchronous local in-process calls, use the platform features (such as Begin and End versions of methods and callbacks) to implement asynchronous method calls.
- Implement asynchronous interfaces as close as possible to the caller to obtain maximum benefit.

- If some recipients can only accept synchronous calls, and you need to support synchronous communication, consider wrapping existing asynchronous calls in a component that performs synchronous communication.

If you choose asynchronous communication and cannot guarantee network connectivity or the availability of the target, consider using a store-and-forward message delivery mechanism to avoid losing messages. When choosing a store-and-forward design strategy:

- Consider using Windows Communications Foundation (WCF) Store-and-Forward transport to safely send messages to Windows Mobile devices regardless of whether the sender or receiver is currently online.
- Consider using local caches to store messages for later delivery in case of system or network interruption.
- Consider using Message Queuing to queue messages for later delivery in case of system or network interruption or failure. Message Queuing can perform transacted message delivery and supports reliable once-only delivery. Message Queuing is available on mobile devices, but note that it has somewhat reduced functionality compared to the desktop version.
- On the mobile infrastructure, consider using BizTalk Server to interoperate with other systems and platforms at enterprise level, or for Electronic Data Interchange (EDI).

Coupling and Cohesion

Communication methods that impose interdependencies between the distributed parts of the application will result in a tightly coupled application. A loosely coupled application uses methods that impose a minimum set of requirements for communication to occur.

When designing for coupling and cohesion, consider the following guidelines:

- For loose coupling, choose a message-based technology such as ASMX or WCF.
- For loose coupling, consider using self-describing data and ubiquitous protocols such as HTTP and SOAP.
- To maintain cohesion, ensure that services and interfaces contain only methods that are closely related in purpose and functional area.

State Management

It may be necessary for the communicating parties in an application to maintain state across multiple requests.

When deciding how to implement state management, consider the following guidelines:

- On a mobile device, consider that state could easily be wiped out by a reset or loss of power. Vital state information should be duplicated or synchronized with the infrastructure supporting the mobile device.
- Only maintain state between calls if it is absolutely necessary, since maintaining state consumes resources and can impact the performance of your application.

- If you are using a state-full programming model within a component or service, consider using a durable data store, such as a database, to store state information and use a token to access the information.
- If you are designing an ASMX service, use the Application Context class to preserve state, since it provides access to the default state stores for application scope and session scope.
- If you are designing a WCF service, consider using the extensible objects that are provided by the platform for state management. These extensible objects allow state to be stored in various scopes such as service host, service instance context and operation context. Note that all of these states are kept in memory and are not durable. If you need durable state, you can use the durable storage (introduced in .NET 3.5) or implement your own custom solution.

Message Format

The format you choose for messages, and the communication synchronicity, affect the ability of participants to exchange data, the integrity of that data, and the performance of the communication channel.

Consider the following guidelines when choosing a message format and handling messages:

- Ensure that type information is not lost during the communication process. Binary serialization preserves type fidelity, which is useful when passing objects between client and server. Default XML serialization serializes only public properties and fields and does not preserve type fidelity.
- Ensure that your application code can detect and manage messages that arrive more than once (idempotency).
- Ensure that your application code can detect and manage multiple messages that arrive out of order (commutativity).

Passing Data Through Tiers - Data Formats

To support a diverse range of business processes and applications, consider the following guidelines when selecting a data format for a communication channel:

- Consider the advantage of using custom objects; these can impose a lower overhead than DataSets and support both binary and XML serialization.
- If your application works mainly with sets of data, and needs functionality such as sorting, searching and data binding, consider using DataSets. Consider that DataSets introduce serialization overhead.
- If you are transferring data from a mobile device running SQL Server CE to the mobile infrastructure servers, consider using merge replication, to conserve memory utilization. This way, data stored in memory will not be duplicated –once in the SQL Server CE store, and once in the DataSet.
- If your application works mainly with instance data, consider using scalar values for better performance.

Data Format Considerations

The most common data formats for passing data across tiers are Scalar values, XML, DataSets, and custom objects. Scalar values will reduce your upfront development costs, however they produce tight coupling that can increase maintenance costs if the value types need to change. XML may require additional up front schema definition but it will result in loose coupling that can reduce future maintenance costs and increase interoperability (for example, if you want to expose your interface to additional XML-compliant callers). DataSets work well for complex data types, especially if they are populated directly from your database. However, it is important to understand that DataSets also contain schema and state information that increases the overall volume of data passed across the network. Custom objects work best when none of the other options meets your requirements, or when you are communicating with components that expect a custom object.

Use the following table to understand the key considerations for choosing a data type.

Type	Considerations
Scalar Values	• You want built-in support for serialization. • You can handle the likelihood of schema changes. Scalar values produce tight coupling that will require method signatures to be modified, thereby affecting the calling code.
XML	• You need loose coupling, where the caller must know about only the data that defines the business entity and the schema that provides metadata for the business entity. • You need to support different types of callers, including third-party clients. • You need built-in support for serialization.
DataSet	• You need support for complex data structures. • You need to handle sets and complex relationships. • You need to track changes to data within the DataSet.
Custom Objects	• You need support for complex data structures. • You are communicating with components that know about the object type. • You want to support binary serialization for performance.

Interoperability Considerations

The main factors that influence interoperability of applications and components are the availability of suitable communication channels, and the formats and protocols that the participants can understand.

Consider the following guidelines for maximizing interoperability:

- To enable communication with wide variety of platforms and devices, consider using standard protocols such as SOAP or REST. With both protocols, the structure of message data is defined using XML schemas.
- Keep in mind that protocol decisions may affect the availability of clients you are targeting. For example, target systems may be protected by firewalls that block some protocols.

- Keep in mind that data format decision may affect interoperability. For example, target systems may not understand platform-specific types, or may have different ways of handling and serializing types.
- Keep in mind that security encryption and decryption decisions may affect interoperability. For example, some message encryption/decryption techniques may not be available on all systems.

Performance Considerations

The design of your communication interfaces and the data formats you use will also have a considerable impact on performance, especially when crossing process or machine boundaries. While other considerations, such as interoperability, may require specific interfaces and data formats, there are techniques you can use to improve performance related to communication between different layers or tiers of your application.

Consider the following guidelines for performance:

- Consider the business and data processing ability of your mobile device. It may be more efficient for the device to pass data to the mobile infrastructure servers, process it and pass back the result.
- Avoid fine-grained "chatty" interfaces for cross-process and cross-machine communication if you know that the systems or devices involved will have reliable connectivity. These require the client to make multiple method calls to perform a single logical unit of work.
- Consider using the Façade pattern to provide a coarse-grained wrapper for existing chatty interfaces. If reliable connectivity is an issue, balance the chatty and coarse-grainedness of your interfaces based on the expected or tested reliability of your connection. Communications dropped in the middle of a message transfer will likely require the entire message be repeated, negating the benefits of a very coarse-grained interface. Smaller messages are more likely to be received in their entirety.
- Use Data Transfer Objects to pass data as a single unit instead of passing individual data types one at a time.
- Reduce the volume of data passed to remote methods where possible. This reduces serialization overhead and network latency.
- If serialization performance is critical for your application, consider using custom classes with binary serialization.
- If XML is required for interoperability, use attribute based structures for large amounts of data instead of element based structures.

Security Considerations

Communication security consists primarily of data protection. A secure communication strategy will protect sensitive data from being read when passed over the network, it will protect sensitive data from being tampered with, and if necessary, it will guarantee the identity of the caller. There are two fundamental areas of concern for securing communications: transport security and message-based security.

Transport Security.

Transport security is used to provide point-to-point security between the two endpoints. Protecting the channel prevents attackers from accessing all messages on the channel. Common approaches to transport security are Secure Sockets Layer (SSL) and IPSec.

Consider the following when deciding to use transport security:

- When using transport security, the transport layer passes user credentials and claims to the recipient.
- Transport security uses common industry standards that provide good interoperability.
- Transport security supports a limited set of credentials and claims compared to message security.
- If interactions between the service and the consumer are not routed through other services, you can use just transport layer security.
- If the message passes through one or more servers, use message-based protection as well as transport layer security. With transport layer security, the message is decrypted and then encrypted at each server it passes through; which represents a security risk.
- Transport security is usually faster for encryption and signing since it is accomplished at lower layers, sometimes even on the network hardware.

Message Security

Message security can be used with any transport protocol. You should protect the content of individual messages passing over the channel whenever they pass outside your own secure network, and even within your network for highly sensitive content. Common approaches to message security are encryption and digital signatures.

Consider the following guidelines for message security:

- Always implement message-based security for sensitive messages that pass out of your secure network.
- Always use message-based security where there are intermediate systems between the client and the service. Intermediate servers will receive the message, handle it, then create a new SSL or IPSec connection, and can therefore access the unprotected message.
- Combine transport and message-based security techniques for maximum protection.

WCF Technology Options

WCF provides a comprehensive mechanism for implementing services in a range of situations, and allows you to exert fine control over the configuration and content of the services. The following guidelines will help you to understand how you can use WCF:

- You can use WCF to communicate with Web services to achieve interoperability with other platforms that also support SOAP, such as the J2EE-based application servers.
- You can use WCF to communicate using SOAP messages and binary encoding for data structures when both the server and the client use WCF.

- You can use WS-Security to implement authentication, data integrity, data privacy, and other security features.
- You can use WS-Reliable Messaging to implement reliable end-to-end communication, even when one or more Web services intermediaries must be traversed.
- You can use WCF to build REST Singleton & Collection Services, ATOM Feed and Publishing Protocol Services, and HTTP Plain XML Services.

WCF supports several different protocols for communication:

- When providing public interfaces that are accessed from the Internet, use the HTTP protocol.
- When providing interfaces that are accessed from within a private network, use the TCP protocol.
- When providing interfaces that are accessed from the same machine, use the named pipes protocol, which supports a shared buffer or streams for passing data.

ASMX Technology Options

ASP.NET Web Services (ASMX) provide a simpler solution for building Web services based on ASP.NET and exposed through an IIS Web server. The following guidelines will help you to understand how you can use ASMX Web services:

- ASMX services can be accessed over the Internet.
- ASMX services use port 80 by default, but this can be easily reconfigured.
- ASMX services support only the HTTP protocol.
- ASMX services have no support for DTC transaction flow. You must program long-running transactions using custom implementations.
- ASMX services support IIS authentication.
- ASMX services support Roles stored as Windows groups for authorization.
- ASMX services support IIS and ASP.NET impersonation.
- ASMX services support SSL transport security.
- ASMX services support the endpoint technology implemented in IIS.
- ASMX services provide cross-platform interoperability and cross-company computing.

REST vs. SOAP

There are two general approaches to the design of service interfaces, and the format of requests sent to services. These approaches are REpresentational State Transfer (REST) and SOAP. The REST approach encompasses a series of network architecture principles that specify target resource and address formats. It effectively means the use of a simple interface that does not require session maintenance or a messaging layer such as SOAP, but instead sends information about the target domain and resource as part of the request URI. The SOAP approach serializes data into an XML format passed as values in an XML message. The XML document is placed into a SOAP envelope that defines the communication parameters such as address, security, and other factors.

When choosing between REST and SOAP, consider the following guidelines:

- SOAP is a protocol that provides a basic messaging framework upon which abstract layers can be built.
- SOAP is commonly used as a remote procedure call (RPC) framework that passes calls and responses over networks using XML-formatted messages.
- SOAP handles issues such as security and addressing through its internal protocol implementation, but requires a SOAP stack to be available.
- REST can be implemented over other protocols, such as JSON and custom Plain Old XML (POX) formats.
- REST exposes an application as a state machine, not just a service endpoint. It has an inherently stateless nature, and allows standard HTTP calls such as GET and PUT to be used to query and modify the state of the system.
- REST gives users the impression that the application is a network of linked resources, as indicated by the URI for each resource.

Chapter 8: Deployment Patterns

Objectives

- Learn the key factors that influence deployment choices.
- Understand the recommendations for choosing a deployment pattern.
- Understand the effect of deployment strategy on performance, security, and other quality attributes.
- Understand the deployment scenarios for each of the application types covered in this guide.
- Learn common deployment patterns.

Overview

Application architecture designs exist as models, documents, and scenarios. However, applications must be deployed into a physical environment where infrastructure limitations may negate some of the architectural decisions. Therefore, you must consider the proposed deployment scenario and the infrastructure as part of your application design process.

This chapter describes the options available for deployment of mobile applications.. By considering the possible deployment scenarios for your application as part of the design process, you prevent a situation where a mobile application with its associated infrastructure cannot be successfully deployed, or fails to perform to its design requirements because of technical infrastructure limitations.

Choosing a Deployment Strategy

Choosing a deployment strategy requires design tradeoffs; for example, because of protocol or port restrictions, or specific deployment topologies in your target environment. Identify your deployment constraints early in the design phase to avoid surprises later. To help you avoid surprises, involve members of your network and infrastructure teams to help with this process.

When choosing a deployment strategy:

- Understand the target physical environment for deployment.
- Understand the architectural and design constraints based on the deployment environment.
- Understand the security and performance impacts of your deployment environment.

Mobile Application Deployment

Mobile applications can be deployed using many different methods. Consider the requirements of your users, as well as how you will manage the application, when designing for deployment. Ensure that you design to allow for the appropriate management, administration, and security for application deployment.

In general, you can deliver your application over the network or manually. Over the network distribution methods will vary by whether your mobile devices are on an intranet or the Internet and what type of infrastructure you have available for delivery. Manual delivery is usually by external storage cards and in rare cases for Original Device Manufacturers (ODMs), use of ROM or firmware memory built into the device itself which holds the application code.

CAB files

If your application will simply be an EXE that runs on the mobile client device, package the EXE in a (.cab) CAB file for distribution. Visual Studio provides a .cab project mechanism to bundle a single or several EXEs in to a .cab file. Consider signing the CAB file to ensure its authenticity, especially if it is being distributed outside a protected network. It is possible to bundle a number of instances of an application into a single .cab file.

You can use several methods to distribute the CAB file.

- Provide an HTML link for the device. Via the device browser, the .cab file can be manually copied onto the device
- If the device can receive e-mail or Short Message Service (SMS) messages, provide a link that the user clicks to download the file via Internet Explorer Mobile.
- Attach the .cab to an e-mail message. The .cab file is installed when the user runs the attachment.
- Use the Application Manager installed with ActiveSync on a host PC to transfer the .cab file on to the device and then install it.
- Provide the .cab on an external memory card. Either include instructions for the user to access and install the .cab file or provide and AutoRun file, assuming that the device is configured to enable Autorun functionality on the device.

Consider the following guidelines when designing your deployment strategy:

- If your users must be able to install and update applications while away from the office, consider designing for over-the-air deployment.
- If you are using CAB file distribution for multiple devices, include multiple device executables in the CAB file. Have the device detect which executable to install, and discard the rest.
- If your application relies heavily on a host PC, consider using ActiveSync to deploy your application.
- If you are deploying a baseline experience running on top of Windows Mobile, consider using the post-load mechanism to automatically load the application after the Windows Mobile operating system starts up.
- If your application will be run only at a specific site, and you want to manually control distribution, consider deployment using an SD memory card

Mobile Application Infrastructure

Mobile devices may or may not require an accompanying infrastructure to support them. The issues to be dealt with are similar to deployment issues for Web Applications that include a service layer. The following information deals with the infrastructure that could help to serve a set of mobile devices and the applications that run on them. Read this section if your devices require servers to provide business services, data synchronization, or deliver installation updates to your mobile devices.

Many of the concepts listed below are the same if you were building an infrastructure for a web application. As such, the concepts are discussed only briefly. Use the companion patterns & practices Web Application Pocket Guide for a future discussion of these architectural concepts.

Distributed vs. Non-distributed Deployment

When creating your deployment strategy for your mobile infrastructure, first determine if you will use a distributed or a non-distributed deployment model. If you are running a web interface on the same set of servers, and you are building a simple application for which you want to minimize the number of required servers, consider a non-distributed deployment. If you do not have another Web UI or are building a more complex application that you will want to optimize for scalability and maintainability, consider a distributed deployment.

Non-distributed Deployment

In a non-distributed deployment, all of the functionality and layers reside on a single server except for data storage functionality, as shown in Figure 1.

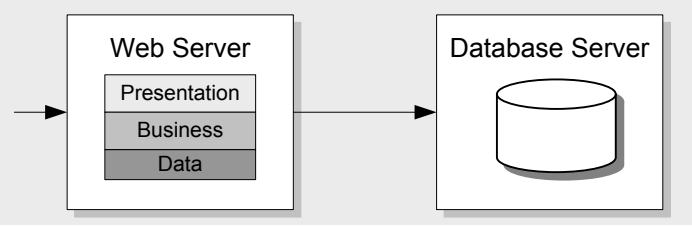

Figure 1 Non-distributed deployment

You would only use this method if you have a web UI in addition to services which would serve your mobile devices. This approach has the advantage of simplicity and minimizes the number of physical servers required. It also minimizes the performance impact inherent when communication between layers has to cross physical boundaries between servers or server clusters. Keep in mind that by using a single server, even though you minimize communication performance overhead, you can hamper performance in other ways. Because all of your layers share resources, one layer can negatively impact all of the other layers when it is under heavy utilization. The use of a single tier reduces your overall scalability and maintainability since all of the layers share the same physical hardware.

Distributed Deployment

In a distributed deployment, the layers of the application reside on separate physical tiers. Distributed deployment allows you to separate the layers of an application on different physical tiers, as shown in Figure 2.

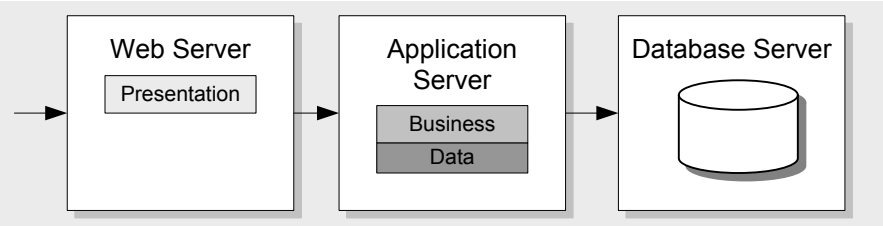

Figure 2 Distributed deployment

If you do not have a Web interface for your mobile application infrastructure, then the Web Server would be removed from the above diagram.

A distributed approach allows you to configure the application servers that host the various layers in order to best meet the requirements of each layer. Distributed deployment also allows you to apply more stringent security to the application servers; for example, by adding a firewall between the Web server and the application servers, and by using different authentication and authorization options. For instance, in a mobile application, the device may use Web services exposed through a Web server, or may access functionality in the application server tier using Distributed COM (DCOM) or Windows Communication Foundation (WCF) services.

Distributed deployment provides a more flexible environment where you can more easily scale out or scale up each physical tier as performance limitations arise, and when processing demands increase.

Performance and Design Considerations for Distributed Environments

Distributing components across physical tiers reduces performance because of the cost of remote calls across server boundaries. However, distributed components can improve scalability opportunities, improve manageability, and reduce costs over time.

Consider the following guidelines when designing an application that will run on a physically distributed infrastructure:

- Choose communication paths and protocols between tiers to ensure that components can securely interact with minimum performance degradation.
- Consider using services and operating system features such as distributed transaction support and authentication that can simplify your design and improve interoperability.
- Reduce the complexity of your component interfaces. Highly granular interfaces (“chatty” interfaces) that require many calls to perform a task work best when located on the same

physical machine. Interfaces that make only one call to accomplish each task ("chunky" interfaces) provide the best performance when the components are distributed across separate physical machines.
- Consider separating long-running critical processes from other processes that might fail by using a separate physical cluster.
- Determine your failover strategy. For example, Web servers typically provide plenty of memory and processing power, but may not have robust storage capabilities (such as RAID mirroring) that can be replaced rapidly in the event of a hardware failure.
- Take advantage of asynchronous calls, one-way calls, or message queuing to minimize blocking when making calls across physical boundaries.
- Determine how best to plan for the addition of extra servers or resources that will increase performance and availability.

Recommendations for Locating Components within a Distributed Deployment

When designing a distributed deployment, you need to determine which layers and components you will put into each physical tier. In most cases you will place the presentation layer on the client or on the Web server; the business, data access, and service layers on the application server; and the database on its own server. In some cases you will want to modify this pattern.

Consider the following guidelines when determining where to locate components in a distributed environment:
- Only distribute components where necessary. Common reasons for implementing distributed deployment include security policies, physical constraints, shared business logic, and scalability.
- In Web applications, deploy business components that are used synchronously by user interfaces (UIs) or user process components in the same physical tier as the UI in order to maximize performance and ease operational management.
- Don't place UI and business components on the same tier if there are security implications that require a trust boundary between them. For instance, you might want to separate business and UI components in a rich client application by placing the UI on the client and the business components on the server.
- Deploy service agent components on the same tier as the code that calls the components, unless there are security implications that require a trust boundary between them.
- Deploy asynchronous business components, workflow components, and business services on a separate physical tier where possible.
- Deploy business entities on the same physical tier as the components that use them.

Scale Up vs. Scale Out

Your approach to scaling is a critical design consideration. Whether you plan to scale out your solution through a Web farm, a load-balanced middle tier, or a partitioned database, you need to ensure that your design supports this.

When you scale your application, you can choose from and combine two basic choices:

- Scale up: Get a bigger box.
- Scale out: Get more boxes.

Scale Up: Get a Bigger Box

With this approach, you add hardware such as processors, RAM, and network interface cards (NICs) to your existing servers to support increased capacity. This is a simple option and can be cost-effective because it does not introduce additional maintenance and support costs. However, any single point of failure remain, which is a risk. Beyond a certain threshold, adding more hardware to the existing servers may not produce the desired results. For an application to scale up effectively, the underlying framework, run time, and computer architecture must scale up as well. When scaling up, consider which resources the application is bound by. If it is memory-bound or network-bound, adding CPU resources will not help.

Scale Out: Get More Boxes

To scale out, you add more servers and use load-balancing and clustering solutions. In addition to handling additional load, the scale-out scenario also protects against hardware failures. If one server fails, there are additional servers in the cluster that can take over the load. For example, you might host multiple Web servers in a Web farm that hosts presentation and business layers, or you might physically partition your application's business logic and use a separately load-balanced middle tier along with a load-balanced front tier hosting the presentation layer. If your application is I/O-constrained and you must support an extremely large database, you might partition your database across multiple database servers. In general, the ability of an application to scale out depends more on its architecture than on underlying infrastructure.

Consider Whether You Need to Support Scale Out

Scaling up with additional processor power and increased memory can be a cost-effective solution. This approach also avoids introducing the additional management cost associated with scaling out and using Web farms and clustering technology. You should look at scale-up options first and conduct performance tests to see whether scaling up your solution meets your defined scalability criteria and supports the necessary number of concurrent users at an acceptable performance level. You should have a scaling plan for your system that tracks its observed growth.

If scaling up your solution does not provide adequate scalability because you reach CPU, I/O, or memory thresholds, you must scale out and introduce additional servers. Consider the following practices in your design to ensure that your application can be scaled out successfully:

- **You need to be able to scale out your bottlenecks, wherever they are**. If the bottlenecks are on a shared resource that cannot be scaled, you have a problem. However, having a class of servers that have affinity with one resource type could be beneficial, but they must then be independently scaled. For example, if you have a single Microsoft SQL Server®

instance that provides a directory, everyone uses it. In this case, when the server becomes a bottleneck, you can scale out and use multiple copies. Creating an affinity between the data in the directory and the SQL Servers that serve the data allows you to concentrate those servers and does not cause scaling problems later, so in this case affinity is a good idea.

- **Define a loosely coupled and layered design**. A loosely coupled, layered design with clean, remotable interfaces is more easily scaled out than tightly-coupled layers with "chatty" interactions. A layered design will have natural clutch points, making it ideal for scaling out at the layer boundaries. The trick is to find the right boundaries. For example, business logic may be more easily relocated to a load-balanced, middle-tier application server farm.

Consider Design Implications and Tradeoffs up Front

You need to consider aspects of scalability that may vary by application layer, tier, or type of data. Know your tradeoffs up front and know where you have flexibility and where you do not. Scaling up and then out with Web or application servers might not be the best approach. For example, although you can have an 8-processor server in this role, economics would probably drive you to a set of smaller servers instead of a few big ones. On the other hand, scaling up and then out might be the right approach for your database servers, depending on the role of the data and how the data is used. Apart from technical and performance considerations, you also need to take into account operational and management implications and related total cost of ownership (TCO) costs.

Stateless Components

If you have stateless components (for example, a Web front end with no in-process state and no stateful business components), this aspect of your design supports both scaling up and scaling out. Typically, you optimize the price and performance within the boundaries of the other constraints you may have. For example, 2-processor Web or application servers may be optimal when you evaluate price and performance compared with 4-processor servers; that is, four 2-processor servers may be better than two 4-processor servers. You also need to consider other constraints, such as the maximum number of servers you can have behind a particular load-balancing infrastructure. In general, there are no design tradeoffs if you adhere to a stateless design. You optimize price, performance, and manageability.

Data

For data, decisions largely depend on the type of data:

- **Static, reference, and read-only data**. For this type of data, you can easily have many replicas in the right places if this helps your performance and scalability. This has minimal impact on design and can be largely driven by optimization considerations. Consolidating several logically separate and independent databases on one database server may or may not be appropriate even if you can do it in terms of capacity. Spreading replicas closer to the consumers of that data may be an equally valid approach. However, be aware that whenever you replicate, you will have a loosely synchronized system.
- **Dynamic (often transient) data that is easily partitioned**. This is data that is relevant to a particular user or session (and if subsequent requests can come to different Web or

application servers, they all need to access it), but the data for user A is not related in any way to the data for user B. For example, shopping carts and session state both fall into this category. This data is slightly more complicated to handle than static, read-only data, but you can still optimize and distribute quite easily. This is because this type of data can be partitioned. There are no dependencies between the groups, down to the individual user level. The important aspect of this data is that you do not query it across partitions. For example, you ask for the contents of user A's shopping cart but do not ask to show all carts that contain a particular item.

- **Core data**. This type of data is well maintained and protected. This is the main case where the "scale up, then out" approach usually applies. Generally, you do not want to hold this type of data in many places because of the complexity of keeping it synchronized. This is the classic case in which you would typically want to scale up as far as you can (ideally, remaining a single logical instance, with proper clustering), and only when this is not enough, consider partitioning and distribution scale-out. Advances in database technology (such as distributed partitioned views) have made partitioning much easier, although you should do so only if you need to. This is rarely because the database is too big, but more often it is driven by other considerations such as who owns the data, geographic distribution, proximity to the consumers, and availability.

Consider Database Partitioning at Design Time

If your application uses a very large database and you anticipate an I/O bottleneck, ensure that you design for database partitioning up front. Moving to a partitioned database later usually results in a significant amount of costly rework and often a complete database redesign.

Partitioning provides several benefits:

- The ability to restrict queries to a single partition, thereby limiting the resource usage to only a fraction of the data.
- The ability to engage multiple partitions, thereby getting more parallelism and superior performance because you can have more disks working to retrieve your data.

Be aware that in some situations, multiple partitions may not be appropriate and could have a negative impact. For example, some operations that use multiple disks could be performed more efficiently with concentrated data. So when you partition, consider the benefits together with alternate approaches.

Performance Patterns

Performance deployment patterns represent proven design solutions to common performance problems. When considering a high-performance deployment, you can scale up or scale out. Scaling up entails improvements to the hardware on which you are already running. Scaling out entails distributing your application across multiple physical servers to distribute the load. A layered application lends itself more easily to being scaled out. Consider the use of Web farms or load-balancing clusters when designing a scale-out strategy.

Web Farms

A *Web farm* is a collection of servers that run the same application. Requests from clients are distributed to each server in the farm, so that each has approximately the same load. Depending on the routing technology used, it may detect failed servers and remove them from the routing list to minimize the impact of a failure. In simple scenarios, the routing may be on a "round robin" basis where a Domain Name System (DNS) server hands out the addresses of individual servers in rotation. Figure 3 illustrates a simple Web farm where each server hosts all of the layers of the application except for the data store.

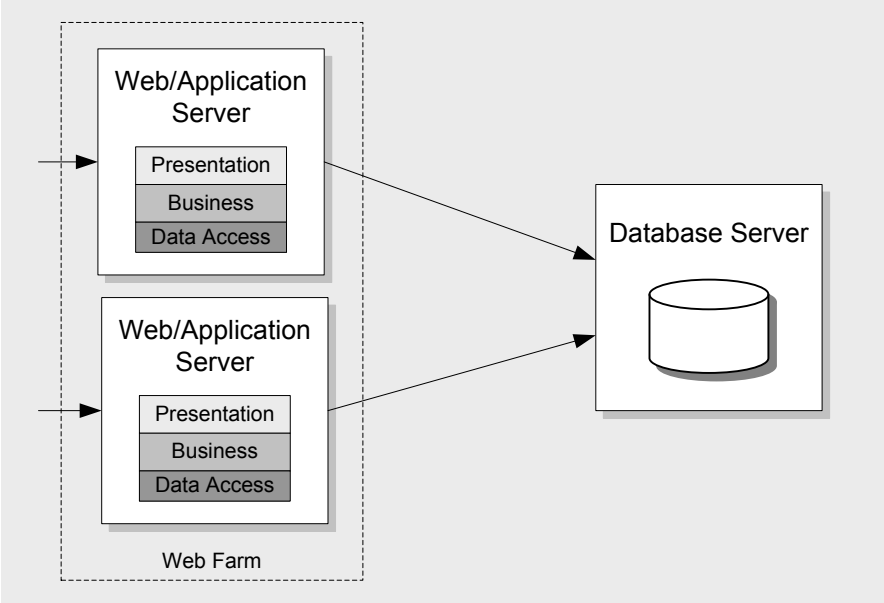

Figure 3 A simple Web farm

Affinity and User Sessions

Web applications often rely on the maintenance of session state between requests from the same user. A Web farm can be configured to route all requests from the same user to the same server—a process known as affinity—in order to maintain state where this is stored in memory on the Web server. However, for maximum performance and reliability, you should use a separate session state store with a Web farm to remove the requirement for affinity.

In ASP.NET, you must also configure all of the Web servers to use a consistent encryption key and method for viewstate encryption where you do not implement affinity. You should also enable affinity for sessions that use Secure Sockets Layer (SSL) encryption, or use a separate cluster for SSL requests.

Application Farms

If you use a distributed model for your application, with the business layer and data layer running on different physical tiers from the presentation layer, you can scale out the business layer and data layer by using an application farm. Requests from the presentation tier are distributed to each server in the farm so that each has approximately the same load. You may decide to separate the business layer components and the data layer components on different

application farms, depending on the requirements of each layer and the expected loading and number of users.

Load-balancing Cluster

You can install your service or application onto multiple servers that are configured to share the workload, as shown in Figure 4. This type of configuration is known as a *load-balanced cluster*.

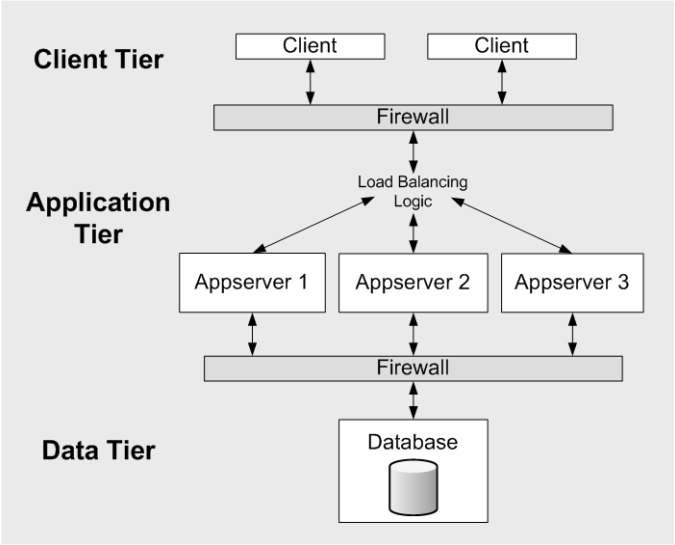

Figure 4 A load-balanced cluster

Load balancing scales the performance of server-based programs, such as a Web server, by distributing client requests across multiple servers. Load-balancing technologies, commonly referred to as load balancers, receive incoming requests and redirect them to a specific host if necessary. The load-balanced hosts concurrently respond to different client requests, even multiple requests from the same client. For example, a Web browser might obtain the multiple images within a single Web page from different hosts in the cluster. This distributes the load, speeds up processing, and shortens the response time to clients.

Reliability Patterns

Reliability deployment patterns represent proven design solutions to common reliability problems. The most common approach to improving the reliability of your deployment is to use a failover cluster to ensure the availability of your application even if a server fails.

Failover Cluster

A *failover cluster* is a set of servers that are configured in such a way that if one server becomes unavailable, another server automatically takes over for the failed server and continues processing. Figure 5 shows a failover cluster.

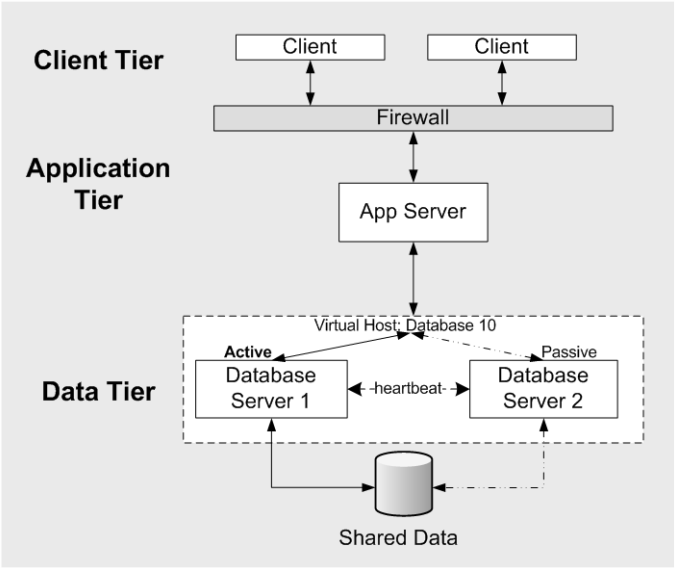

Figure 5 A failover cluster

Install your application or service on multiple servers that are configured to take over for one another when a failure occurs. The process of one server taking over for a failed server is commonly known as *failover*. Each server in the cluster has at least one other server in the cluster identified as its standby server.

Security Patterns

Security patterns represent proven design solutions to common security problems. The impersonation/delegation approach is a good solution when you must flow the context of the original caller to downstream layers or components in your application. The trusted subsystem approach is a good solution when you want to handle authentication and authorization in upstream components and access a downstream resource with a single trusted identity.

Impersonation/Delegation

In the impersonation/delegation authorization model, resources and the types of operations (such as read, write, and delete) permitted for each one are secured using Windows Access Control Lists (ACLs) or the equivalent security features of the targeted resource (such as tables and procedures in SQL Server). Users access the resources using their original identity through impersonation, as illustrated in Figure 6.

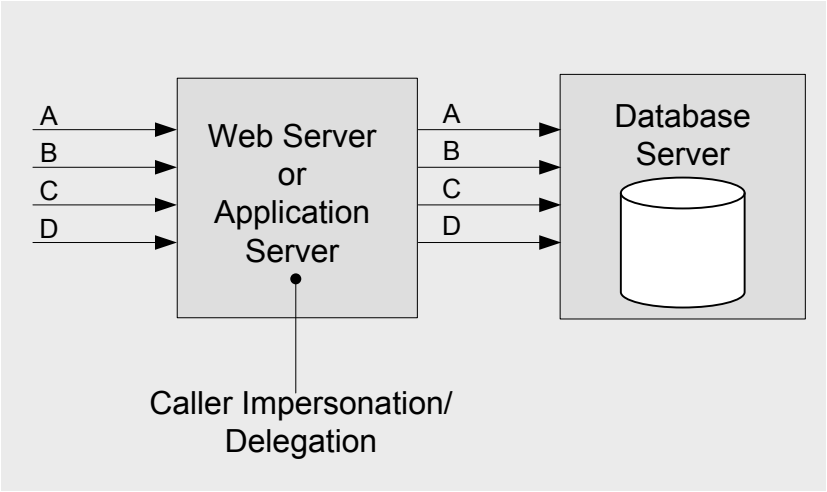

Figure 6 The impersonation/delegation authorization model

Trusted Subsystem

In the trusted subsystem (or trusted server) model, users are partitioned into application-defined, logical roles. Members of a particular role share the same privileges within the application. Access to operations (typically expressed by method calls) is authorized based on the role membership of the caller. With this role-based (or operations-based) approach to security, access to operations (not back-end resources) is authorized based on the role membership of the caller. Roles, analyzed and defined at application design time, are used as logical containers that group together users who share the same security privileges or capabilities within the application. The middle-tier service uses a fixed identity to access downstream services and resources, as illustrated in Figure 7.

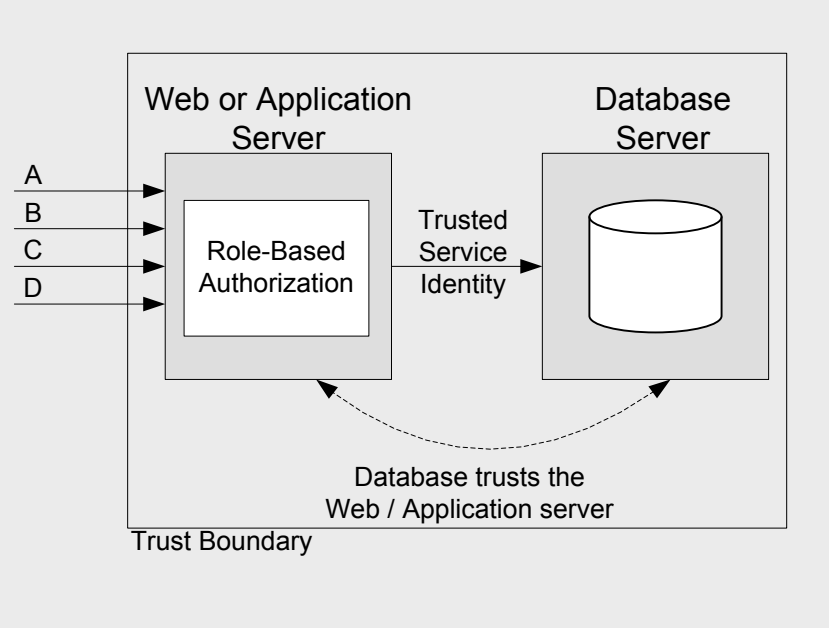

Figure 7 The trusted subsystem (or trusted server) model

Multiple Trusted Service Identities

In some situations, you might require more than one trusted identity. For example, you might have two groups of users, one who should be authorized to perform read/write operations and the other read-only operations. The use of multiple trusted service identities provides the ability to exert more granular control over resource access and auditing, without having a large impact on scalability. Figure 8 illustrates the multiple trusted service identities model.

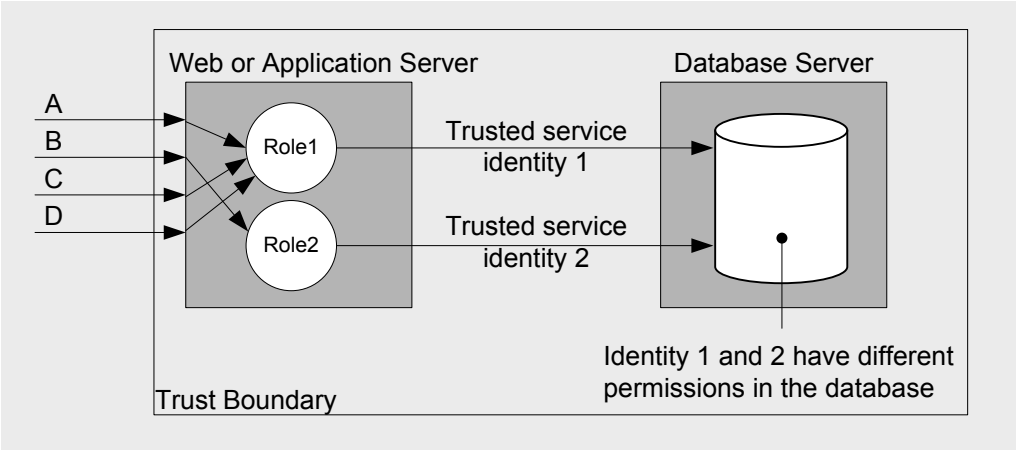

Figure 8 The multiple trusted service identities model

Network Infrastructure Security Considerations

Make sure that you understand the network structure provided by your target environment, and understand the baseline security requirements of the network in terms of filtering rules, port restrictions, supported protocols, and so on. Recommendations for maximizing network security include:

- Identify how firewalls and firewall policies are likely to affect your application's design and deployment. Firewalls should be used to separate the Internet-facing applications from the internal network, and to protect the database servers. These can limit the available communication ports and, therefore, authentication options from the Web server to remote application and database servers. For example, Windows authentication requires additional ports.
- Consider what protocols, ports, and services are allowed to access internal resources from the Web servers in the perimeter network or from rich client applications. Identify the protocols and ports that the application design requires, and analyze the potential threats that occur from opening new ports or using new protocols.
- Communicate and record any assumptions made about network and application layer security, and what security functions each component will handle. This prevents security controls from being missed when both development and network teams assume that the other team is addressing the issue.
- Pay attention to the security defenses that your application relies upon the network to provide, and ensure that these defenses are in place.

- Consider the implications of a change in network configuration, and how this will affect security.

Manageability Considerations

The choices you make when deploying an application affect the capabilities for managing and monitoring the application. You should take into account the following recommendations:

- Deploy components of the application that are used by multiple consumers in a single central location to avoid duplication.
- Ensure that data is stored in a location where backup and restore facilities can access it.
- Components that rely on existing software or hardware (such as a proprietary network that can only be established from a particular computer) must be physically located on the same computer.
- Some libraries and adaptors cannot be deployed freely without incurring extra cost, or may be charged on a per-CPU basis; therefore, you should centralize these features.
- Groups within an organization may own a particular service, component, or application that they need to manage locally.
- Monitoring tools such as System Center Operations Manager require access to physical machines to obtain management information, and this may impact deployment options.
- The use of management and monitoring technologies such as Windows Management Instrumentation (WMI) may impact deployment options.

Pattern Map

Key patterns are organized by key categories such as Deployment, Manageability, Performance & Reliability, and Security in the following table. Consider using these patterns when making design decisions for each category.

Category	Relevant patterns
Deployment	• Layered Application • Three-Layered Services Application • Tiered Distribution • Three-Tiered Distribution • Deployment Plan
Manageability	• Adapter • Provider
Performance & Reliability	• Server Clustering • Load-Balanced Cluster • Failover Cluster
Security	• Brokered Authentication • Direct Authentication • Federated Authentication (SSO) • Impersonation and Delegation • Trusted Subsystem

- For more information on the Layered Application, Three-Layered Services Application, Tiered Distribution, Three-Tiered Distribution, and Deployment Plan patterns, see "Deployment Patterns" at http://msdn.microsoft.com/en-us/library/ms998478.aspx
- For more information on the Server Clustering, Load-Balanced Cluster, and Failover Cluster patterns, see "Performance and Reliability Patterns" at http://msdn.microsoft.com/en-us/library/ms998503.aspx
- For more information on the Brokered Authentication, Direct Authentication, Impersonation and Delegation, and Trusted Subsystem patterns, see "Web Service Security" at http://msdn.microsoft.com/en-us/library/aa480545.aspx
- For more information on the Provider pattern, see "Provider Model Design Pattern and Specification, Part 1" at http://msdn.microsoft.com/en-us/library/ms972319.aspx
- For more information on the Adapter pattern, see "data & object factory" at http://www.dofactory.com/Patterns/Patterns.aspx

Key Patterns

- **Adapter**. An object that supports a common interface and translates operations between the common interface and other objects that implement similar functionality with different interfaces.
- **Brokered Authentication**. A pattern that authenticates against a broker, which provides a token to use for authentication when accessing services or systems.
- **Direct Authentication**. A pattern that authenticates directly against the service or system that is being accessed.
- **Layered Application**. An architectural pattern where a system is organized into layers.
- **Load-Balanced Cluster**. A distribution pattern where multiple servers are configured to share the workload. Load balancing provides both improvements in performance by spreading the work across multiple servers, and reliability where one server can fail and the others will continue to handle the workload.
- **Provider**. A pattern that implements a component that exposes an API that is different from the client API, in order to allow any custom implementation to be seamlessly plugged in. Many applications that provide instrumentation expose providers that can be used to capture information about the state and health of your application and the system hosting the application.
- **Tiered Distribution**. An architectural pattern where the layers of a design can be distributed across physical boundaries.
- **Trusted Subsystem**. A pattern where the application acts as a trusted subsystem to access additional resources. It uses its own credentials instead of the user's credentials to access the resource.

Additional Resources

- For more information on deploying applications to Mobile devices, see http://msdn.microsoft.com/en-us/library/bb158580.aspx.

- For more information on authorization techniques, see *Designing Application-Managed Authorization* at http://msdn.microsoft.com/en-us/library/ms954586.aspx.
- For more information on deployment scenarios and considerations, see *Deploying .NET Framework-based Applications* at http://msdn.microsoft.com/en-us/library/ms954585.aspx.
- For more information on design patterns, see *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn.microsoft.com/en-us/library/ms998469.aspx.

Cheat Sheet: Presentation Technology Matrix

Objectives

- Understand the tradeoffs for each presentation technology choice.
- Understand the design impact of choosing a presentation technology.
- Understand all presentation technologies across application types.
- Choose a presentation technology for your scenario and application type.

Overview

Use this cheat sheet to understand your technology choices for the presentation layer. Your choice of presentation technology will be related to both the application type you are developing and the type of user experience you plan to deliver. Use the Presentation Layer Technology Summary to review each technology and its description. Use the Benefits and Considerations Matrix to make an informed choice of presentation technology based on the advantages and considerations of each one. Use the Common Scenarios and Solutions to map your application scenario to common presentation technology solutions.

Presentation Technologies Summary

Mobile Applications

The following presentation technologies are suitable for use in mobile applications:

- **Microsoft .NET Compact Framework**. This is a subset of the Microsoft .NET Framework designed specifically for mobile devices. Use this technology for mobile applications that must run on the device without guaranteed network connectivity.
- **ASP.NET Mobile**. This is a subset of ASP.NET, designed specifically for mobile devices. ASP.NET Mobile applications can be hosted on a normal ASP.NET server. Use this technology for mobile Web applications when you need to support a large number of mobile devices and browsers that can rely on a guaranteed network connection.
- **Microsoft® Silverlight® Mobile**. This subset of the Silverlight client requires the Silverlight plug-in to be installed on the mobile device. Use this technology to port existing Silverlight applications to mobile devices, or if you want to create a richer user interface (UI) than is possible using other technologies.

Note: At the time this guide was published, Silverlight for Mobile was an announced product under development, but not yet released.

Benefits and Considerations Matrix

Mobile Applications

Technology	Benefits	Considerations
.NET Compact Framework	• Runs on the client machine for improved performance and responsiveness • Does not require 100-percent network connectivity. • Has a familiar programming model if you are used to Windows Forms. • Has Microsoft Visual Studio® Designer support. • Is usually installed in ROM on the device.	• Has a limited API compared to a desktop Windows Forms application. • Requires more client-side resources than an ASP.NET Mobile application. • Is not as easy to deploy over the Web as an ASP.NET Mobile application.
ASP.NET Mobile	• Supports a wide range of devices, including anything that has a Web browser. • Does not have a footprint on the device because no application must be installed. • Has a familiar programming model if you are used to ASP.NET Web Forms. • Templates can be downloaded off the Web for designer support with Visual Studio.	• Design support has been removed from Visual Studio 2008, but the controls will still render on devices. • Requires 100-percent network connectivity to run. • Performance and responsiveness are dependent on network bandwidth and latency. • Many devices now support full HTML and ASP.NET, so ASP.NET Mobile may not be required.
Silverlight Mobile	• Offers rich UI and visualization, including 2-D graphics, vector graphics, and animation. • Silverlight code running on desktops can run on Silverlight Mobile. • Isolated storage is available to maintain objects outside the browser cache.	• Is an announced product in development, but not yet released at the time this document was published. • Uses more device resources than a Web application. • Desktop Silverlight applications running on mobile may require optimization to account for reduced memory and slower hardware. • Requires the Silverlight plug-in to be installed. • May not run on as many types of devices as Web applications because of plug-in install requirement.

Common Scenarios and Solutions

Mobile Applications

.NET Compact Framework

Consider using the .NET Compact Framework if:

- You are building a mobile application that must support occasionally connected or offline scenarios.
- You are building a mobile application that will run on the client to maximize performance and responsiveness.

ASP.NET for Mobile

Consider using ASP.NET for Mobile if:

- You are building an application that can rely on 100-percent network connectivity.
- Your team has ASP.NET expertise and you want to target a wide range of mobile devices.
- You are building an application that must have no client-side installation or plug-in dependencies.
- You want to support the widest possible range of devices.
- You want to use as few device resources as possible.

Silverlight for Mobile

Consider using Silverlight for Mobile if:

- You are building a Web application and want to leverage the rich visualization and UI capabilities of Silverlight.
- The devices you are targeting have easy access to or already have the Silverlight plug-in installed.

Cheat Sheet: Data Access Technology Matrix

Objectives

- Understand the tradeoffs for each data-access technology choice on the Windows Mobile platform.
- Understand the design impact of choosing a data-access technology.
- Understand all data-access technologies across application types.
- Choose a data-access technology for your scenario and application type.

Overview

Use this cheat sheet to understand your technology choices for the data access layer. Your choice of data-access technology will be related both to the application type you are developing and the type of business entities you choose for your data layer. Use the Data Access Technologies Summary to review each technology and its description. Use the Benefits and Considerations Matrix to make an informed choice of data-access technology based on the advantages and considerations for each one. Use the Common Scenarios and Solutions section to map your application scenarios to common data-access technology solutions.

Mobile Device Considerations

As mentioned through this document, a number of technologies are not available on the Windows Mobile operating system, but may be useful for synchronization/service interfaces running on a PC or server.

The following technologies are not available on Windows Mobile at the time of publication:

- ADO.NET Entity Framework
- ADO.NET Data Services Framework
- LINQ to Entities
- LINQ to SQL
- LINQ to Data Services
- ADO.NET Core; Windows Mobile supports only SQL Server and SQL Server Compact Edition

Be sure to check the product documentation to verify availability for later versions.

Data Access Technologies Summary

The following data-access technologies are available with the Microsoft .NET platform:

- **ADO.NET**. ADO.NET provides general retrieval, update, and management of data. On Windows Mobile, ADO.NET includes providers for Microsoft® SQL Server®, and Microsoft® SQL Server® Compact Edition.
- **ADO.NET Sync Services**. ADO.NET Sync Services is a provider included in the Microsoft Sync Framework synchronization for ADO.NET-enabled databases. It enables data

synchronization to be built in occasionally connected applications. It periodically gathers information from the client database and synchronizes it with the server database.

- **Language-Integrated Query (LINQ)**. LINQ provides class libraries that extend C# and Microsoft Visual Basic® with native language syntax for queries. Queries can be performed against a variety of data formats, including DataSet (LINQ to DataSet), XML (LINQ to XML), in-memory objects (LINQ to Objects),.Understand that LINQ is primarily a query technology supported by different assemblies throughout the .NET Framework. For example, LINQ to Entities is included with the ADO.NET Entity Framework assemblies; LINQ to XML is included with the System.Xml assemblies; and LINQ to Objects is included with the .NET core system assemblies.

Benefits and Considerations Matrix

Disconnected and Offline

Technology	Benefits	Considerations
LINQ to DataSet	• Allows full-featured queries against a DataSet.	• Processing is all on the client side.
ADO.NET Sync Services	• Enables synchronization between databases, collaboration, and offline scenarios. • Synchronization can execute in the background. • Provides a hub-and-spoke type of architecture for collaboration between databases.	• Change tracking ability needs to be provided. • Exchanging large chunks of data during synchronization can reduce performance.
ADO.NET	• Includes .NET managed code providers for connected access to a wide range of data stores. • Provides facilities for disconnected data storage and manipulation.	• Only a subset of the ADO.NET Core. Does not support System.Data.OleDB namespace. • Code is written directly against specific providers, thereby reducing reusability. • The relational database structure may not match the object model, requiring you to write a data mapping layer by hand.
LINQ to Objects	• Allows you to create LINQ-based queries against objects in memory. • Represents a new approach to retrieving data from collections. • Can be used directly with any collections that support IEnumerable or IEnumerable<T>. • Can be used to query strings, reflection-based metadata, and file directories.	• Will only work with objects that implement the IEnumerable interface.

Technology	Benefits	Considerations
LINQ to XML	• Allows you to create LINQ-based queries against XML data. • Comparable to the Document Object Model (DOM), which brings an XML document into memory, but is much easier to use. • Query results can be used as parameters to XElement and XAttribute object constructors.	• Relies heavily on generic classes. • Is not optimized to work with untrusted XML documents, which require different mitigation techniques for security.

General Recommendations

Consider the following general recommendations:

- **Microsoft Windows Mobile® Devices**. Many data technologies are not supported on Windows Mobile Devices because they are too heavy considering the limited memory capabilities. For on the device data management, primarily utilize SQL Server Compact Edition and ADO.NET Sync Services to maintain data on a mobile device and synchronize it with a larger database system. Features such as merge replication can also assist in Windows Mobile scenarios.
- **Flexibility and performance on the Server**. If you need maximum performance and flexibility on a server supporting a mobile device, consider using ADO.NET. ADO.NET provides the most capabilities and is the most server-specific solution. When using ADO.NET, consider the tradeoff of additional flexibility versus the need to write custom code. Keep in mind that mapping to custom objects will reduce performance.
  Offline/Occasionally connected scenarios. If you must support a disco**nnected scenario, consider usin**g **DataSets.** Utilize the the Sync Framework or SQL Server merge replication if you have will have at least an initial connection to put the required data onto the mobile device.

Note: You might need to mix and match the data-access technology options for your scenario. Start with what you need.

Common Scenarios and Solutions

ADO.NET

Consider using ADO.NET if you:

- Are on a mobile device and want to access data through SQL Server Compact Edition.
- Need to use low-level APIs for full control over data access in your application.
- Are building an application that needs to support a disconnected data-access experience.

ADO.NET Sync Services

Consider using ADO.NET Sync Services if you:

- Need to build an application that supports occasionally connected scenarios.

- Need collaboration between databases.
- Are using Windows Mobile and want to sync with a central database server.

LINQ to DataSets

Consider using LINQ to DataSets if you:

- Want to execute queries against a Dataset, including queries that join tables.
- Want to use a common query language instead of writing iterative code.

LINQ to Objects

Consider using LINQ to Objects if you:

- Need to execute queries against a collection.
- Want to execute queries against file directories.
- Want to execute queries against in-memory objects using the LINQ syntax.

LINQ to XML

Consider using LINQ to XML if you:

- Are using XML data in your application.
- Want to execute queries against XML data using the LINQ syntax.

Additional Resources

For more information, see the following resources:

- *ADO.NET* at http://msdn.microsoft.com/en-us/library/e80y5yhx(vs.80).aspx.
- *Language-Integrated Query (LINQ)* at http://msdn.microsoft.com/en-us/library/bb397926.aspx.
- *SQL Server Data Services (SSDS) Primer* at http://msdn.microsoft.com/en-us/library/cc512417.aspx.
- *Introduction to the Microsoft Sync Framework Runtime at http://msdn.microsoft.com/en-us/sync/bb821992.aspx*

Checklist – Mobile Application

Design Considerations

- You have determined the device types you will support.
- You have designed the application with occasionally-connected limited-bandwidth scenarios in mind.
- The user interface design is appropriate for the mobile device.
- You did not try to reuse a desktop application design or user interface.
- You have considered device resource constraints for battery life, memory size and program speed.

Authentication and Authorization

- You have designed authentication for over-the-air, cradled synchronization scenarios, Bluetooth discovery, and local SD card scenarios.
- You have considered that different devices may have variations in their programming security model, which can affect authorization to access resources
- You did not assume that security mechanisms available on larger platforms will be available on a mobile platform.
- You have designed authentication for access to Bluetooth devices.
- Trust boundaries are identified within your mobile application layers.

Caching

- Performance objectives are identified, such as minimum response time and battery life.
- You are caching static data that is useful, and avoided caching volatile data except when required for an offline scenario.
- You have considered data caching requirements for occasionally-connected scenarios.
- Your design supports low memory detection, and prioritization of data to discard, as available memory decreases.
- You chose the appropriate cache location, such as on the device, at the mobile gateway, or in the database server.

Communication

- Your design supports asynchronous, threaded communication to improve usability in occasionally-connected scenarios.
- You have considered the effects of receiving a phone call during communication or program execution.
- Communication over un-trusted connections, such as Web services and other over-the-air methods, is protected.
- You have considered using Web services for communication when you must access data from multiple sources, interoperate with other applications, or work while disconnected.

- You have considered message queuing when you need guaranteed delivery, asynchronous messaging, message security, or transaction support.

Configuration Management

- Your design supports the saving and restoration of configuration data after a device reset.
- Merge replication with a "buy and configure" application from a third party is used, if the enterprise data is in Microsoft SQL Server 2005 or 2008 and desire an accelerated time to market.
- You are using a binary format instead of XML for configuration files due to memory limitations.
- You are protecting sensitive data in device configuration files.
- You have considered using compression library routines to reduce the memory requirements for configuration and state information.
- You have considered using the Mobile Device Manager interface to manage group configuration of devices when you have an Active Directory infrastructure.

Data Access

- You have designed application to ensure data integrity.
- You are using the device database services (such as SQL Server Compact Edition) when your application must access a disconnected database.
- You are using merge replication to synchronize large volumes of data in one operation over a high bandwidth network connection.
- You have considered the overall size and impact on performance when you use XML to store or transfer data.
- You are using custom strongly-typed or generic data transfer objects instead of DataSets to reduce memory overhead and improve performance.
- You are using the Microsoft Sync Framework when you must synchronize individual sets of a data over a remote connection.

Debugging

- You understand debugging costs associated with devices you need to support.
- You are using an emulator for initial testing and debugging.
- You are using the device for final testing.
- You have tested fully-disconnected scenarios in the design.
- You understand that it may be difficult to maintain context between different types of code running on a device.

Deployment

- Your design supports over-the-air deployment when users must be able to install and update applications while away from the office.
- You are using ActiveSync to deploy your application when the application relies heavily on a host PC.

- You are using the post-load mechanism to automatically load your application immediately after the Windows Mobile OS starts up when you are deploying a baseline experience running on top of Windows Mobile.
- When targeting multiple devices, you have included multiple device executables in the CAB when you are using CAB file distribution.
- Application will be deployed using an SD memory card if the application will be run only at a specific site and you want to manually control distribution.

Device

- The application is optimized for the device by considering factors such as screen size and orientation, network bandwidth, memory and storage space, and other hardware capabilities.
- Device-specific capabilities have been considered that will enhance your application functionality, such as accelerometers, GPUs, GPS, haptic (touch, force and vibration) feedback, compass, camera and fingerprint readers.
- You have designed a core functionality sub set when you are developing for more than one device.
- You have considered adding customization for device-specific features, including functionality to detect when the code is running on a device that can utilize this functionality.
- You have created modular code that can be removed if separate executable files are required due to device memory size constraints.

Exception Management

- Your application is designed to recover to a known good state after an exception.
- You did not use exceptions to control logic flow.
- Sensitive information in exception messages and log files are not revealed to users.
- Unhandled exceptions are dealt with appropriately.
- You have designed an appropriate logging and notification strategy for critical errors and exceptions.

Logging

- You did not store sensitive information in log files.
- You have considered using the Mobile Device Manager to extract logs from mobile devices when you have an Active Directory infrastructure.
- You have considered using platform features such as health monitoring on the server, and mobile device services on the device, to log and audit events.
- You have considered logging in an abbreviated or compressed format to minimize memory and storage impact when you carry out extensive logging on the device.
- You have used OpenNetCF on the device if you do not require extensive logging.
- You have decided what constitutes unusual or suspicious activity on a device, and logged information based on these scenarios.

Porting

- The application was re-written in its entirety when porting a Rich Client application from the desktop.
- The user interface has been redesigned for the smaller screen size when porting a Web application to a mobile device.
- You have researched details to discover what code will port without modification when porting a RIA client.
- You have researched and utilized tools to assist in porting.
- You have tested the custom controls which when porting to a mobile application and based on the results decided to rewrite to find an alternative.

Power

- The user interface is not updated while the application is in the background.
- Communication methods that use the least amount of power necessary are used.
- Power profiles have been implemented to increase performance when device is plugged into external power and not charging the battery.
- Power consumption has been considered when using the device CPU, wireless communication, screen, or other power-consuming resources while on battery power.
- Device functionality is allowed to be powered down when not in use or not needed. Common examples are screen backlighting, hard drives, GPS functions, speakers, wireless communications.

Synchronization

- Your design supports recovery when synchronization is reset, and is able to manage synchronization conflicts.
- You have considered the Microsoft Sync Framework when you want easy-to-implement one-way synchronization.
- You have considered using merge replication synchronization when you must support bidirectional synchronization to SQL Server.
- You have considered including over-the-air synchronization in your design when users must synchronize data when away from the office.
- You have considered including cradled synchronization in your design when users will be synchronizing with a host PC.
- You have considered store and forward synchronization using WCF over email or SMS (text message) mechanisms.

User Interface (UI)

- You have provided the user with a visual indication of blocking operations; for example, an hourglass cursor.
- You did not place menu bars at the top of the screen as they are difficult to see when using a stylus or touch screen input.

- The design of your layout considers input from various sources. For example, making buttons large enough for devices that supports touch screen.
- Your design supports various screen sizes and orientations associated with devices you need to support.

Performance Considerations

- Your design supports configurable options to allow the maximum use of device capabilities.
- You have considered using lazy initialization to optimize for mobile device resource constraints.
- You have considered limited memory resources and optimized your application to use the minimum amount of memory.
- You have considered using programming shortcuts as opposed to following pure programming practices that can inflate code size and memory consumption.
- You have balanced performance requirements with power consumption requirements.

Checklist - Presentation Layer

Design Considerations

- Relevant presentation layer patterns are identified and used in the design; for example, use the Template View pattern for dynamic Web pages.
- The application is designed to separate rendering components from components that manage presentation data and process.
- Organizational UI guidelines are well understood and addressed by the design.
- The design is based upon knowledge of how the user wants to interact with the system.

Caching

- Volatile data is not cached, unless caching data required in a disconnected scenario.
- Sensitive data is not cached unless absolutely necessary and encrypted.
- Data is cached is a ready to use format to reduce processing after the cached data is retrieved.
- An in-memory cache is used unless the cache must be stored persistently.
- Your design includes a strategy for expiration, scavenging and flushing; for example, scavenging based on absolute expiration if it is in-memory and you can predict the time at which the data will change.
- The caching strategy has been tested to see if it improves performance

Exception Management

- Sensitive data or internal application details are not revealed to users in error messages or in exceptions.
- User-friendly error messages are displayed in the event of an exception that impacts the user.
- Unhandled exceptions are captured.
- Exceptions are not used to control application logic.
- The set of exceptions that can be thrown by each component is well understood.
- Exceptions are logged to support troubleshooting when necessary.

Input

- Form-based input is used for normal data collection.
- Wizard-based input is used for complex data collection tasks or input that requires workflow.
- Device-dependant input, such as ink or speech, is considered in the design.
- Accessibility was considered in the design.
- Localization was considered in the design.

Layout

- Templates are used to provide a common look and feel.
- A common layout is used to maximize accessibility and ease of use.
- User personalization is considered in the layout design.

Navigation

- Navigation is separated from UI processing.
- If access to navigation state is required across sessions, the application is designed to persist navigation state.
- If navigation logic is complex, the UI is decoupled from the navigation logic.
- The Page Controller pattern is used to separate business logic from the presentation logic.
- The Front Controller pattern is used to configure complex page navigation logic dynamically.

Request Processing

- Requests do not block the UI.
- Long running requests are identified in the design and optimized for UI responsiveness.
- UI request processing uses unique components that are not mixed with components that render the UI, or with components that instantiate business rules.

User Experience

- Error messages are designed with the target user in mind.
- UI responsiveness is considered in the design; for example, rich clients do not block UI threads and Rich Internet Applications avoid synchronous processing.
- The design has identified key user scenarios and has made them as simple to accomplish as possible.
- The design empowers users, allowing them to control how they interact with the application and how it displays data.

UI Components

- Platform provided controls are used except for where it is absolutely necessary to use a custom or third-party control for specialized display or input tasks.
- Platform provided databinding is used where possible.
- State is stored in the user's session for ASP.NET Mobile Web applications.

UI Processing Components

- If the UI requires complex processing, UI processing has been decoupled from rendering and display into unique UI processing components.
- If the UI requires complex workflow support, the design includes unique workflow components that use a workflow system such as Windows Workflow.
- UI processing has been divided into model, view and controller or presenter by using the MVC pattern.

- UI processing components do not include business rules.

Validation

- The application constrains, rejects and sanitizes all input that comes from the client.
- Built-in validation controls are used when possible.
- Validation routines are centralized, where possible, to improve maintainability and reuse.

Checklist - Business Layer

Design Considerations

- A separate business layer is used to implement the business logic and workflows.
- Component types are not mixed in the business layer.
- Common business logic functions are centralized and reused.
- Business layer is not tightly coupled to other layers.

Business Components

- Business components do not mix data access logic and business logic.
- Components are designed to be highly cohesive.
- Business components are invoked with message-based communication.
- All processes exposed through the service interfaces are idempotent.
- Workflow components are used, if the business process involves multiple steps and long-running transactions.

Business Entities

- Appropriate data format is used to represent business entities.
- The Table Module pattern is used to design business entities, if the tables in the database represent the business entities.
- Business entities support serialization if they need to be passed over network or stored directly to the disk.
- The Data Transfer Object (DTO) pattern is used to minimize the number of calls made across tiers.

Caching

- Static data that will be regularly reused within the business layer is cached.
- Data that cannot be retrieved from the database quickly and efficiently is cached.
- Data is cached in ready-to-use format.
- Sensitive data is not cached.

Coupling and Cohesion

- The design does not require tight coupling between the business layer and other layers.
- A message-based interface is used for the business layer.
- The business layer components are highly cohesive.
- Data access logic is not mixed with business logic in the business components.

Concurrency and Transactions

- Business critical operations are wrapped in transactions.
- Connection-based transactions are used when accessing a single data source.

- The design defines transaction boundaries, so that retries and composition are possible.
- A compensating method to revert the data store to its previous state is used, when transactions are not possible.
- Locks are not held during long-running atomic transactions, compensating locks are used instead.
- Appropriate transaction isolation level is used.

Data Access

- Data access code and business logic are not mixed with the business components.
- The business layer does not directly access the database; instead, a separate data access layer is used.

Exception Management

- Exceptions are not used to control business logic.
- Exceptions are caught only if they can be handled
- Appropriate exception propagation strategy is designed.
- Global error handler is used to catch unhandled exceptions.
- The design includes a notification strategy for critical errors and exceptions.
- Exceptions do not reveal sensitive information.

Logging and Instrumentation

- Logging and instrumentation solution is centralized for the business layer.
- System-critical and business-critical events in your business components are logged.
- Access to the business layer is logged.
- Business-sensitive information is not written to log files.
- Logging failure does not impact normal business layer functionality.

Validation

- All input is validated in the business layer, even when input validation occurs in the presentation layer.
- The validation solution is centralized for reusability.
- Validation strategy constrains, rejects, and sanitizes malicious input.
- Input data is validated for length, format, and type.

Workflow

- Workflows are used within business components that involve multi-step or long-running processes.
- Appropriate workflow style is used depending on the application scenario.
- The workflow fault conditions are handled appropriately.
- The Pipeline pattern is used, if the component must execute a specified set of steps sequentially and synchronously.

- The Event pattern is used, if the process steps can be executed asynchronously in any order.

Checklist - Data Access Layer

Design Considerations

- Abstraction is used to implement a loosely coupled interface to the data access layer.
- Data access functionality is encapsulated within the data access layer.
- Application entities are mapped to data source structures.
- Data exceptions that can be handled are caught and processed.
- Connection information is protected from unauthorized access.

Blob

- Images are stored in a database only when it is not practical to store them on the file system.
- BLOBs are used to simplify synchronization of large binary objects between servers.
- Additional database fields are used to provide query support for BLOB data.
- BLOB data is cast to the appropriate type for manipulation within your business or presentation layer.

Batching

- Batched commands are used to reduce round trips to the database and minimize network traffic.
- Largely similar queries are batched for maximum benefit.
- Batched commands are used with a DataReader to load or copy multiple sets of data.
- Bulk copy utilities are used when loading large amounts of file-based data into the database.
- You do not place locks on long running batch commands.

Connections

- You are maintaining a single connection throughout the lifetime of the application, with SQL Server CE.
- Transactions are performed through a single connection when possible.
- You do not rely on garbage collection to free connections.
- Retry logic is used to manage situations where the connection to the data source is lost or times out.

Data Format

- You have considered the use of custom data or business entities for improved application maintainability.
- Business rules are not implemented in data structures associated with the data layer.
- XML is used for structured data that changes over time.

- DataSets have been considered for disconnected operations when dealing with small amounts of data.
- Serialization and interoperability requirements have been considered.

Exception Management

- You have identified data access exceptions that should be handled in the data layer.
- Global exception handling has been implemented to catch unhandled exceptions.
- Data source information has been included when logging exceptions and errors.
- Sensitive information in exception messages and log files is not revealed to users.
- You have designed an appropriate logging and notification strategy for critical errors and exceptions.

Queries

- Parameterized SQL statements are used, instead of assembling statements from literal strings, to protect against SQL Injection attacks.
- User input has been validated when used with dynamically generated SQL queries.
- String concatenation has not been used to build dynamic queries in the data layer.
- Objects are used to build the database query.

Transactions

- Transactions are enabled only when actually required.
- Transactions are kept as short as possible to minimize the amount of time that locks are held.
- Manual or explicit transactions are used when performing transactions against a single database.
- You are not using nested transactions with SQL Server CE.
- You are not using distributed transactions with .NET Compact Framework.

Validation

- All data received by the data layer is validated.
- User input used to dynamic SQL has been validated to protect against SQL injection attacks.
- You have determined whether validation that occurs in other layers is sufficient, or if you must validate it again.
- The data layer returns informative error messages if validation fails.

XML

- XML readers and writers are used to access XML-formatted data.
- Custom validators are used for complex data parameters within your XML schema.
- XML indexes have been considered for read-heavy applications that use XML in SQL Server.

Manageability Considerations

- A common interface or abstraction layer is used to provide an interface to the data layer.
- You have considered creating custom entities, or if other data representations better meet your requirements.
- Business or data entities are defined by deriving them from a base class that provides basic functionality and encapsulates common tasks.
- Business or data entities rely on data access logic components for database interaction.

Performance Considerations

- Isolation levels have been tuned for data queries.
- Commands are batched to reduce round-trips to the database server.
- Optimistic concurrency is used with non-volatile data to mitigate the cost of locking data in the database.
- Ordinal lookups are used for faster performance when using a DataReader.

Security Considerations

- Windows authentication has been used when available.
- Encrypted connection strings in configuration files are used instead of a System name.
- A salted hash is used instead of an encrypted version of the password when storing passwords.
- Identity information is passed to the data layer for auditing purposes.
- Typed parameters are used with stored procedures and dynamic SQL to protect against SQL injection attacks.

Checklist - Service Layer

Design Considerations

- Services are designed to be application scoped and not component scoped.
- Entities used by the service are extensible and composed from standard elements.
- Your design does not assume to know who the client is.
- Your design assumes the possibility of invalid requests.
- Your design separates functional business concerns from infrastructure operational concerns.

Authentication

- Your have identified a suitable mechanism for securely authenticating users.
- Your have considered the implications of using different trust settings for executing service code.
- SSL protocol is used if you are using basic authentication.
- WS Security is used if you are using SOAP messages.

Authorization

- Appropriate access permissions are set on resources for users, groups, and roles.
- URL authorization and/or file authorization is used appropriately if you are using Windows authentication.
- Access to Web methods is restricted appropriately using declarative principle permission demands.
- Services are run using least privileged account.

Communication

- You have determined how to handle unreliable or intermittent communication scenarios.
- Dynamic URL behavior is used to configure endpoints for maximum flexibility.
- Endpoint addresses in messages are validated.
- You have determined the approach for handling asynchronous calls.
- You have decided if the message communication must be one-way or two-way.

Data Consistency

- All parameters passed to the service components are validated.
- All input is validated for malicious content.
- Appropriate signing, encryption, and encoding strategies are used for protecting your message.
- XML schemas are used to validate incoming SOAP messages.

Exception Management

- Exceptions are not used to control business logic.
- Unhandled exceptions are dealt with appropriately.
- Sensitive information in exception messages and log files is not revealed to users.
- SOAP Fault elements or custom extensions are used to return exception details to the caller when using SOAP.
- Tracing and debug-mode compilation for all services is disabled except during development and testing.

Message Channels

- Appropriate patterns, such as Channel Adapter, Messaging Bus, and Messaging Bridge are used for messaging channels.
- You have determined how you will intercept and inspect the data between endpoints when necessary.

Message Construction

- Appropriate patterns, such as Command, Document, Event, and Request-Reply are used for message constructions.
- Very large quantities of data are divided into relatively smaller chunks and sent in sequence.
- Expiration information is included in time-sensitive messages, and the service ignores expired messages.

Message Endpoint

- Appropriate patterns such as Gateway, Mapper, Competing Consumers, and Message Dispatcher are used for message endpoints.
- You have determined if you should accept all messages, or implement a filter to handle specific messages.
- Your interface is designed for idempotency so that, if it receives duplicate messages from the same consumer, it will handle only one.
- Your interface is designed for commutativity so that, if messages arrive out of order, they will be stored and then processed in the correct order.
- Your interface is designed for disconnected scenarios, such as providing support for guaranteed delivery.

Message Protection

- The service is using transport layer security when interactions between the service and consumer are not routed through other servers.
- The service is using message-based protection when interactions between the service and consumer are routed through other servers.
- You have considered message-based plus transport layer (mixed) security when you need additional security.

- Encryption is used to protect sensitive data in messages.
- Digital signatures are used to protect messages and parameters from tampering.

Message Routing

- Appropriate patterns such as Aggregator, Content-Based Router, Dynamic Router, and Message Filter are used for message routing.
- The router ensures sequential messages sent by a client are all delivered to the same endpoint in the required order (commutativity).
- The router has access to the message information when it needs to use that information for determining how to route the message.

Message Transformation

- Appropriate patterns such as Canonical Data Mapper, Envelope Wrapper, and Normalizer are used for message transformation.
- Metadata is used to define the message format.
- An external repository is used to store the metadata when appropriate.

Representational State Transfer (REST)

- You have identified and categorized resources that will be available to clients.
- You have chosen an approach for resource identification that uses meaningful names for REST starting points and unique identifiers, such as a GUID, for specific resource instances.
- You have decided if multiple views should be supported for different resources, such as support for GET and POST operations for a specific resource.

Service Interface

- A coarse-grained interface is used to minimize the number of calls.
- The interface is decoupled from the implementation of the service.
- Business rules are not included in the service interface.
- The schema exposed by the interface is based on standards for maximum compatibility with different clients.
- The interface is designed without assumptions about how the service will be used by clients.

SOAP

- You have defined the schema for operations that can be performed by a service.
- You have defined the schema for data structures passed with a service request.
- You have defined the schema for errors or faults that can be returned from a service request.

Deployment Considerations

- The service layer is deployed to the same tier as the business layer in order to maximize service performance.

- You are using Named Pipes or Shared Memory protocols when a service is located on the same physical tier as the service consumer.
- You are using the TCP protocol when a service is accessed only by other applications within a local network.
- You are using the HTTP protocol when a service is publicly accessible from the Internet.