Microsoft

Rich Client Architecture Guide

Application Architecture Pocket Guide Series

patterns & practices

Rich Client 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 Rich Client Application Architecture Pocket Guide is to improve your effectiveness when building Rich Client 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 Rich Client 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, "Rich Client Application Architecture,"** provides general design guidelines for a Rich Client 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 Rich Client 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 service 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

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 – Rich Client Application Architecture

Objectives

- Define a Rich Client Application.
- Understand key scenarios where Rich Client applications would be used.
- Understand components found in a Rich Client application.
- Learn about design considerations.
- Understand deployment scenarios for Rich Client applications.
- Learn the key patterns and technology considerations.

Overview

Rich Client user interfaces can provide high performance, interactive, and rich user experiences for applications that must operate in stand-alone, connected, occasionally connected, and disconnected scenarios. Windows Forms, WPF, and Office Business Application (OBA) development environments and tools are available that allow developers to quickly and easily build Rich Client applications.

While these technologies can be used to create standalone applications, they can also be used to create applications that run on the client machine, but communicate with services exposed by other layers (both logical and physical) that expose operations the client requires. These operations may include data access, information retrieval, searching, sending information to other systems, backup, and related activities.

Figure 1 shows an overall view of a typical Rich Client architecture, and identifies the components usually found in each layer.

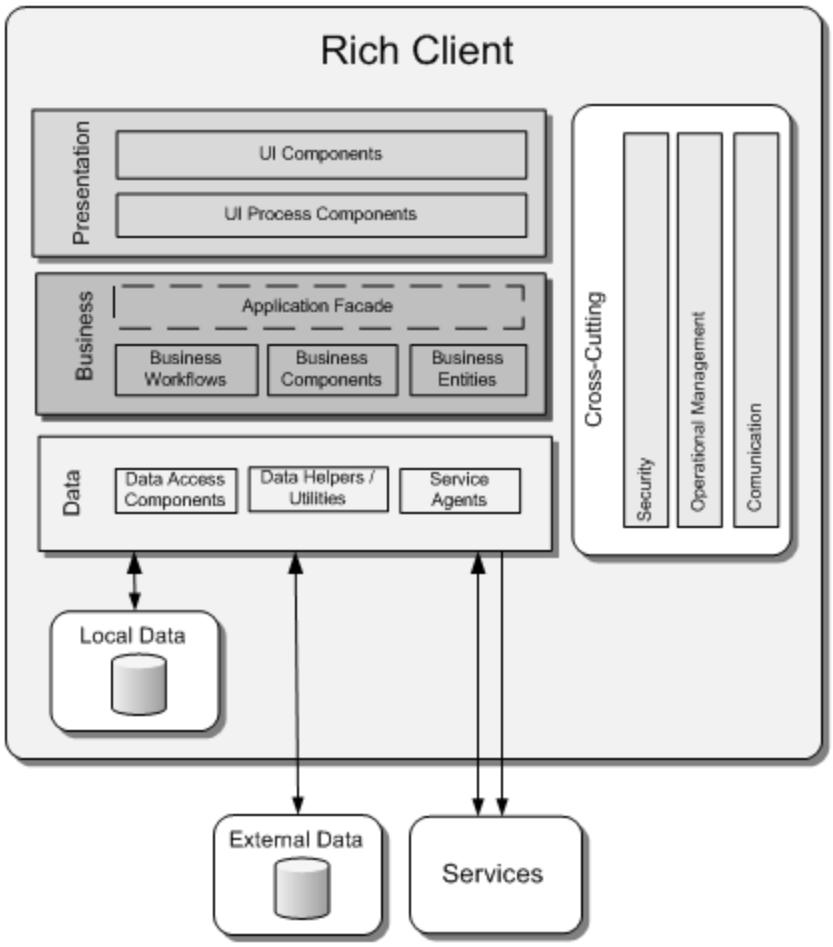

Figure 1 - An overall view of a typical Rich Client architecture.

Key Scenarios

Rich Client applications can range from fairly thin interfaces that overlay business layers and service layers, to complex applications that perform most of the processes themselves and just communicate with other layers to consume or send back information. An example is the difference between an application such as an FTP client that depends on a remote server to provide all of the data for display, and an application like Microsoft Excel that performs complex local tasks, stores state and data locally, and only communicates with remote servers to fetch and update linked data.

Therefore, the design and implementation of a Rich Client varies a great deal. However, in presentation layer and User Interface (UI) terms, there are some common approaches to good architectural design. Most of these are based on well-known design patterns. Many of these patterns encourage the use of separate components within the application that reduce dependencies, make maintenance and testing easier, and promote reusability.

Some Rich Client applications are effectively stand-alone, and rely only on data retrieved and sent back to other layers of the application and to other applications. These types of Rich Client may contain their own business layers and data access layers. They may also expose business and data services for other applications to use. The guidelines for the business and data layers in such applications are the same as those discussed generally for all applications.

Key Components

A Rich Client application generally contains presentation layer components, which include UI components and UI processing components; business layer components, which include business workflow, business processing, business entity components and, optionally, a façade; and data layer components, which include data access, data helper/utility, and service agent components. The following list describes each of the component types in more detail:

- **User interface (UI) components**. User interface components provide a way for users to interact with the application. They render and format data for display to users, and acquire and validate data that users enter.
- **User process components**. To help synchronize and orchestrate these user interactions, it can be useful to drive the process using separate user process components. This avoids hard coding the process flow and state management logic in the user interface elements themselves, and the same basic user interaction patterns can be reused by multiple user interfaces.
- **Business components**. Business components implement the business logic of the application. Regardless of whether a business process consists of a single step or an orchestrated workflow, your application will probably require components that implement business rules and perform business tasks.
- **Business workflows**. After the required data is collected by a user process, the data can be used to perform a business process. 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 they can be implemented using business process management tools.
- **Business entity components**: Business entities are used to pass data between components. The data represents real-world business entities, such as products or orders. The business entities that are used internally in the application are usually data structures, such as DataSets, DataReaders, or Extensible Markup Language (XML) streams, but they can be implemented as custom object-oriented classes that represent the real-world entities your application has to work with, such as a product or an order.
- **Application Facade**. (Optional). A façade is used to combine multiple business operations into single messaged based operation. You might access the application façade from the presentation layer using a range of communication technologies.
- **Data access logic components**. Data access components abstract the logic necessary to access your underlying data stores. Doing so centralizes data access functionality and makes it easier to configure and maintain.
- **Data Helpers/Utilities**. Implement data helpers for centralizing generic data access functionality such as managing database connections and caching data. You can design data

source-specific helper components to abstract the complexity of accessing the database. Avoid adding any business logic to the helper components.

- **Service agents**. When a business component must use functionality provided in an external service, you may need to provide some code to manage the semantics of communicating with that particular service. Service agents isolate the idiosyncrasies of calling diverse services from your application, 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.

Design Considerations

When designing a Rich Client application, the goal of a software architect is to choose an appropriate technology and design a structure that minimizes the complexity by separating tasks into different areas of concern. The design should meet the requirements for the application in terms of performance, security, reusability, and ease of maintenance. When designing Rich Client applications, consider the following guidelines:

- **Choose an appropriate technology based on application requirements**. Technologies include Windows Forms, Windows Presentation Foundation (WPF), XAML Browser Applications (XBAP), and Office Business applications (OBA).
- **Separate presentation logic from interface implementation**. Design patterns such as MVC and Supervising Controller separate the UI rendering from UI processing, which eases maintenance, promotes reusability, and improves testability.
- **Identify the presentation tasks and presentation flows**. This will help you to design each screen, and each step in multi-screen or Wizard processes.
- **Design to provide a suitable and usable interface**. Take into account features such as layout, navigation, choice of controls, and localization to maximize accessibility and usability.
- **Extract business rules and other tasks not related to the interface**. A separate business layer should handle tasks not directly related to presentation, and not related to collecting and handling user input.
- **Reuse common presentation logic**. Libraries that contain templates, generalized client-side validation functions, and helper classes may be reusable in several applications.
- **Loosely couple your client from remote services it uses**. Use a message-based interface to communicate with services located on separate physical tiers.
- **Avoid tight coupling to objects in other layers**. Use the abstraction provided by common interface definitions, abstract base classes, or messaging when communicating with other layers of the application. For example, implementing the Dependency Injection and Inversion of Control patterns can provide a shared abstraction between layers.
- **Reduce round trips when accessing remote layers**. Use coarse-grained methods and execute them asynchronously if possible to avoid blocking or freezing the user interface.

Rich 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.

Category	Key Issues
Communication	• Choosing the wrong communication protocols and technology. • Using synchronous communication methods when asynchronous methods could provide better interface performance. • Failing to properly detect and manage disconnected or occasionally connected scenarios. • Using fine-grained "chatty" interfaces across physical tiers.
Composition	• Choosing an inappropriate composition technology. • Not managing auto-update and versioning of composable components. • Failing to take advantage of appropriate templates and data-binding technologies. • Failing to take into account personalization requirements.
Configuration Management	• Failing to manage configuration information correctly. • Not securing sensitive configuration information. • Failing to identify the appropriate configuration options and information. • Failing to take into account Group Policy overrides.
Data Services	• Failing to design support for the appropriate data format, such as custom objects, Table Module, Domain Model, Data Transfer Objects, or DataSets. • Failing to manage offline data access, concurrency, and subsequent synchronization correctly. • Not minimizing data returned from remote services and layers. • Failing to support large sets of data when required.
Exception Management	• Not planning a strategy for handling and managing exceptions. • Raising exceptions when not appropriate. • Failing to sanitize exceptions and remove sensitive information. • Not implementing appropriate logging and auditing functionality. • Failing to support administrative and remote management and error-reporting requirements.
State Management	• Failing to store and manage UI state information correctly. • Failing to cache state where this is appropriate. • Choosing an inappropriate cache store. • Failing to protect and secure sensitive state information. • Failing to support transactions where required.

Workflow	• Failing to design and implement an appropriate workflow or viewflow mechanism. • Not implementing error and exception management for workflows and viewflows.

Communication

Rich Clients can communicate with other layers of the application and with other services using a variety of protocols and methods. These may include HTTP requests, SMTP email messaging, SOAP Web service messages, SNTP time synchronization packets, DCOM for remote components, and many other TCP/IP-based standard or custom communication protocols. Alternatively, if the client application is located on the same physical tier as the business layer, then you should use object-based interfaces to interact with the business layer.

When designing a communication strategy, consider the following guidelines:

- When communicating with business layers, services, and components on a remote physical tier, use a message-based protocol when possible. This gives you a more natural way to make asynchronous calls to avoid locking the presentation layer and support load-balanced and failover server configurations.
- When communicating with business layers, services, and components on a remote physical tier, use coarse-grained interfaces to minimize network traffic and maximize performance.
- Where practical, enable offline processing for the application. Detect connection state. When disconnected, cache information locally and then re-synchronize when communication is re-enabled. Consider holding application state and data locally in a persistent cache to allow disconnected start ups and a shutdown / restart without information loss.
- To protect sensitive information and communication channels, consider using IPSec and SSL to secure the channel, encryption to protect data and digital signatures to detect data tampering.
- If the application must consume or send large sets or amounts of data consider the performance and network impact. Choose more efficient communication protocols like TCP, using compression mechanisms to minimize the data payload size for message-based protocols such as SMTP and SOAP, or custom binary formats when the application does not have need to support open communication standards.

Composition

Complex user interfaces are common in business applications. Users may open several forms to perform specific tasks, and work with data in a range of different ways. To maximize extensibility and maintainability of the application, consider implementing the interface using the Composition design pattern, where the UI consists of separate modules or forms loaded dynamically at runtime. Users can open and close forms as required, and the application can maximize performance and reduce start-up delays by loading these forms only when required. Also, consider how you can support personalization for users, so that they can modify the layout and content to suit their own requirements.

When designing a composition strategy, consider the following guidelines:

- Based on functional specifications and requirements, identify the appropriate types of interface components you require. For example, possible components include Windows Forms, WPF Forms, Office-style documents, user controls, or custom modules.
- Be careful with dependencies between components. Use abstraction patterns when possible to avoid issues with maintainability.
- Identify an appropriate composition mechanism. You may decide to use a composition framework or built-in features of your development environment such as user controls or document panels.
- Consider composing views from reusable modular parts. For example use the Composite View pattern to build a view from modular, atomic component parts.
- If you need to allow communication between presentation components, consider implementing the Publish/Subscribe pattern. This will lower the coupling between the components and improve testability.

Configuration Management

Rich Client applications will usually require configuration information loaded at start-up, and sometimes during execution. This information may be network or connection information, user settings, part of the UI business rules, or general display and layout settings. You may decide to store some or all of this information locally, or download it from other layers when the application starts. You may also need to persist changes to the information as the application runs or when it ends; for example, storing user preferences, layout settings, and other UI data in a user's local profile.

When designing a configuration management strategy, consider the following guidelines:

- Determine what configurable data may change in the life of your application. For example, file locations, developer vs. production settings, logging, assembly references, and contact information for notifications.
- Choose local or centralized storage locations. User managed data should be stored locally. Global application settings should be in a central location and perhaps downloaded locally for performance reasons.
- Identify sensitive configuration information, and implement a suitable mechanism for protecting it during transit over the network, when persisted locally, and even when stored in memory.
- Take into account any global security policies that may affect or override local configurations.

Data Services

Rich Client applications will usually access data stored on a remote physical tier, as well as data stored on local physical layers and even on the local machine. Data access often has a significant impact on performance, and is the most obvious factor in the user's perception of an application and its usability and responsiveness. You should aim to maximize performance of

data access routines and data transmission across tiers. You must also design the application with regard to the types of data it will use, and the way that this data is exposed from other layers of the application.

When designing a data services strategy, consider the following guidelines:

- If the client application cannot handle the data in the exposed format you must implement a translation mechanism that converts it, but this will have an impact on performance.
- If the client will consume very large amounts of data, consider chunking these and loading them asynchronously into a local cache to improve performance. You will have to handle inconsistencies between the local copy and the original data using methods such as time-stamps or events.
- Whenever possible, load data asynchronously so that the user interface is still responsive while the data is loading. However, you must also be aware of conflicts that may occur if the user attempts to interact with the data before loading is complete, and design the interface to protect against errors arising from this.
- In occasionally connected scenarios, monitor connectivity and implement a service dispatcher mechanism to support batch processing to allow users to perform multiple updates to data.
- Determine how you will detect and manage concurrency conflicts that arise when multiple users attempt to update the data store. Explore optimistic and pessimistic concurrency models.

Exception Management

All applications and services are subject to the occurrence of errors and exceptions, and you must implement a suitable strategy for detecting and managing these errors and exceptions. In a Rich Client application, you will usually need to notify the user. In addition, for anything other than trivial UI errors such as validation messages, you should implement a mechanism that logs errors and exceptions, notifies administrators, and exposes information for use by management tools and monitoring systems.

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

- Identify the errors and exceptions likely to arise within the application, and identify which of these require only user notification. Errors such as validation failures are usually only notified locally to the user. However, errors such as repeated invalid logon attempts or detection of malicious data should be logged and administrators notified. All execution exceptions and application failures should be logged and administrators notified.
- Identify an overall strategy for handling exceptions. This may involve actions such as wrapping exceptions with other application-specific or custom exceptions that contain additional data to assist in resolving failures, or replacing exceptions to prevent exposure of sensitive information. Also, implement a mechanism for detecting unhandled exceptions and logging these. A framework for managing exceptions may be useful for these tasks.
- Determine how you will store exception information, how you will pass it to other layers of the application if required, and how you will notify administrators. Consider using a

monitoring tool or environment that can read events from the local machine and present a view of the application state to administrators.
- Ensure that you sanitize exception information that is exposed to users to prevent sensitive information being displayed or stored in log and audit files. If necessary, encrypt information and use secure channels to communicate exceptions and errors to other physical tiers of the application.

State Management

Rich Clients, whether designed to run offline or only when connected, will generally store state information. This may include user settings, configuration information, workflow information, business rule values, and data that the UI is displaying and the user is editing. The application must be able to save this data, access it as required, and handle conflicts, restarts, and connection status changes.

When designing a state management strategy, consider the following guidelines:
- Determine the state information that the application must cache, including estimates of the size, the frequency of changes, and the processing or overhead cost of recreating or re-fetching the data. These factors will help you to decide what type of state mechanism to use.
- If you have large volumes of data, consider using a local disk-based mechanism.
- If the application requires data to be available when it starts up, use a persistent mechanism such as Isolated Storage or a disk file.
- When storing sensitive data, ensure that you implement the appropriate level of protection by using encryption and/or digital signatures.
- Identify state that is applicable to the whole application, and state that is applicable to individual users or all users in a specific role.

Workflow

Some Rich Client applications require viewflow or workflow support to enable multi-stage operations or Wizard-style UI elements. You can implement these features using separate components, custom solutions, or take advantage of a framework such as Windows Workflow (WF) or, for document-based interfaces, Microsoft Office SharePoint Server (MOSS).

When designing a workflow strategy, consider the following guidelines:
- Use workflow within business components for operations that involve multi-step or long-running processes.
- For simple workflow and viewflow requirements, it is usually sufficient to use custom code based on well-known patterns such as Use Case Controller and ViewFlow.
- For more complex workflow and viewflow requirements, consider using a platform provided workflow engine such as Windows Workflow (WF).
- Consider creating separate components to implement your workflow and viewflow tasks. This reduces dependencies and makes it easier to swap out components as requirements change.

- Consider how you will capture, manage, and display errors in workflows.
- Identify how you will handle part-completed tasks, and whether it is possible to recover from a failure and continue the task or whether you need to restart the process.

Presentation Considerations

Rich Client applications often implement the presentation layer for business applications. They are the part of the application that the user sees and interacts with, and must therefore satisfy many requirements. These requirements encompass general factors such as usability, performance, design, and interactivity. A poor user interface can have a negative impact on a business application that performs well in all other aspects.

When designing the presentation features of you application, consider the following guidelines:

- Investigate how you can separate data used by the UI, which may be cached or stored locally, from the UI itself. This makes it easier to update parts of the application, allows developers and designers to work separately on the components, and improves testability.
- Take advantage of data binding capabilities to display data whenever possible, especially for tabular and multi-row data presentation. This reduces the code required, simplifies development, and reduces coding errors. It can also automatically synchronize data in different views or forms. Use two-way binding where the user must be able to update the data.
- Consider how you will display documents in an Office document-style interface, or when displaying document content or HTML in other UI elements. Ensure that the user is protected from invalid and malicious content that may reside in documents.
- Implement command and navigation strategies and mechanisms that are flexible and can be updated easily. Consider implementing well-known design patterns such as Command, Publish/Subscribe, and Observer to decouple commands and navigation from the components in the application and improve testability.
- Ensure that the application can be globalized and localized to all geographical and cultural scenarios where it may be used. This includes changing the language, text direction, and content layout based on configuration or auto-detection of the user's culture.

Business/Service Layer Considerations

Rich Client applications will, in most business scenarios, access data or information located outside the application. While the nature of the information will vary, it is likely to be extracted from a business system. A Rich Client application may act as the presentation layer for a business application; or may include a business layer, data access layer, and service agents that communicate with remote services. You should design the Rich Client interface using the accepted principles for designing presentation layers. In addition, to maximize performance and usability, you may consider locating some of the business processing tasks on the client.

When designing interaction with business and service layers, consider the following guidelines:

- Identify the business layers and service interfaces that the application will use. Import the interface definitions and write code that accesses the business layer service functions using

the interfaces. This helps to minimize coupling between the client and the business layer and services.
- If your business logic does not contain sensitive information, consider locating some of the business rules on the client to improve performance of the UI and the client application.
- If your business logic does contain sensitive information, you should locate the business layer on a separate tier.
- Consider how the client will obtain information required to operate business rules and other client-side processing, and how it will update the business rules automatically as requirements change. You may wish to have the client obtain business rule information from the business layer when it starts up.

Maintainability Considerations

It is vital to minimize maintenance cost and effort for all applications and components. Rich Client applications are usually located remotely from the main servers of an application, and are subsequently more difficult to maintain than server-installed services and components. You should implement mechanisms that reduce maintenance liabilities. The issues to consider include deployment, updates, patches, and versioning.

When designing a maintainability strategy, consider the following guidelines:
- Implement a suitable mechanism for manual and/or automatic updates to the application and its components. You must take into account versioning issues to ensure that the application has consistent and interoperable versions of all the components it uses.
- Choose an appropriate deployment approach based on the environment your application will be used. For example, you may need an installation program for applications that are publically available or you may be able to use system tools to deploy applications within a closed environment.
- Design the application so that components are interchangeable where possible, allowing you to change individual components depending on requirements, runtime scenarios, and individual user requirements or preferences.
- Implement logging and auditing as appropriate for the application to assist administrators and developers when debugging the application and solving runtime problems.
- Design to minimize dependencies between components and layers so that the application can be used in different scenarios where appropriate, and to reduce the possibility of changes to other layers affecting the client application.

Security Considerations

Security encompasses a range of factors and is vital in all types of applications. Rich Client applications must be designed and implemented to maximize security, and – where they act as the presentation layer for business applications – must play their part in protecting and securing the other layers of the application. Security issues involve a range of concerns, including protecting sensitive data, user authentication and authorization, guarding against attack from malicious code and users, and auditing and logging events and user activity.

When designing a security strategy, consider the following guidelines:

- Determine the appropriate technology and approach for authenticating users, including support for multiple users of the same Rich Client application instance. You should consider how and when to log on users, whether you need to support different types of users (different roles) with differing permissions (such as administrators and standard users), and how you will record successful and failed logons. Take into account the requirements for disconnected or offline authentication where this is relevant.
- Consider a Single-Sign-On (SSO) or federated authentication solution if users must be able to access multiple applications with the same credentials or identity. You can implement a suitable solution by registering with an external agency that offers federated authentication, use certificate systems, or create a custom solution for your organization.
- Consider the need to validate inputs, both from the user and from sources such as services and other application interfaces. You may need to create custom validation mechanisms, or you may be able to take advantage of validation frameworks. Visual Studio Windows Forms development environment contains validation controls. The Enterprise Library Validation Application Block provides comprehensive features for validation in the UI and in the business layer. Irrespective of your validation choice, remember that you must always validate data when it arrives at other layers of the application.
- Consider how you will protect data stored in the application and in resources such as files, caches, and documents used by the application. Encrypt sensitive data where it may be exposed, and consider using a digital signature to prevent tampering. In maximum security applications, consider encrypting volatile information stored in memory. Also, remember to protect sensitive information that is sent from the application over a network or communication channel.
- Consider how you will implement auditing and logging for the application, and what information to include in these logs. Remember to protect sensitive information in the logs using encryption, and optionally use digital signatures for the most sensitive types of information that is vulnerable to tampering.

Data Handling Considerations

Application data can be made available from server-side applications through a Web service. Cache this data on client to improve performance and enable offline usage. Rich Client applications can also use local data.

Types of Data

Data uses by Rich Client applications falls into two categories:

- **Read-only reference data**. Data that is not changed by the client, and is used by the client for reference purposes. Store reference data on the client to reduce the amount of data interchange between the client and the server to improve the performance of your application, enable offline capabilities, provide early data validation, and generally improve the usability of your application. In cases where the client does change the data for local purposes, there is no need to keep track of client-side changes to the data on the server.

- **Transient data**. Data that is changed on the client as well as the on server. One of the most challenging aspects of dealing with transient data in Rich Client applications is that it can generally be modified by multiple clients at the same time. You must keep track of any client-side changes made to transient data on the client.

Caching Data

Rich Clients often need to cache data locally, whether it is read-only reference data or transient data. Caching data can improve performance in your application and provide the data necessary to work offline. To enable data caching, Rich Client applications should implement some form of caching infrastructure that can handle the data caching details transparently. The common types of caching are:

- **Short-term data caching**: Data is not persistent, so the application cannot run offline.
- **Long-term data caching**: Caching data in a persistent medium, such as isolated storage or the local file system, allows the application to work when there is no connectivity to the server. Rich Client applications should differentiate between data that has been successfully synchronized with the server and data that is still tentative.

Data Concurrency

Changes to the data held on the server can occur before any client-side changes are synchronized with the server. You must implement a mechanism to ensure that any data conflicts are handled appropriately when the data is synchronized, and the resulting data is consistent and correct.

Approaches for handling data concurrency are:

- **Pessimistic concurrency.** Pessimistic concurrency allows one client to maintain a lock over the data to prevent any other clients from modifying the data until the client's own changes are complete.
- **Optimistic concurrency.** Optimistic concurrency does not lock the data. The original data is then checked against the current data to see if it has been updated since last retrieved.

Using ADO.NET DataSets to Manage Data

The ADO.NET DataSet helps clients to work with data while offline. They can keep track of local changes made to the data, which makes it easier to synchronize the data with the server and reconcile data conflicts. Data Sets can also be used to merge data from different sources.

Windows Forms Data Binding

Windows Forms data binding supports bi-directional binding that allows you to bind a data structure to a user interface component, display the current data values to the user, allow the user to edit the data, and then automatically update the underlying data using the values entered by the user.

Data binding can be used to:

- Display read-only data to users.

- Allow users to update data within the user interface.
- Provide master-detail views of data.
- Allow users to explore complex related data items.
- Provide lookup table functionality, allowing the user interface to display user-friendly names for data items instead of data row key values.

Offline/Occasionally Connected Considerations

An application is occasionally connected if, during unspecified periods, it cannot interact with services or data over a network in a timely manner. Occasionally connected Rich Client applications are capable of performing work when not connected to a networked resource, and can update the networked resources in the background when a connection is available.

Offline/Occasionally Connected Design Strategies:

Consider the following two approaches when designing for an occasionally connected scenario:

- **Data-centric**. Applications that use the data-centric strategy have a relational database management system (RDBMS) installed locally on the client, and use the built-in capabilities of the database system to propagate local data changes back to the server, handle the synchronization process, and detect and resolve any data conflicts.
- **Service-oriented**. Applications that use the service-oriented approach store information in messages, and arrange these messages in queues while the client is offline. After the connection is re-established, the queued messages are sent to the server for processing.

Principles of Occasionally Connected Applications

Consider the following guidelines for designing occasionally connected applications:

- Favor asynchronous communication when interacting with data and services over a network.
- Minimize or eliminate complex interactions with network-located data and services.
- Add data caching capabilities. Ensure that all of the data necessary for the user to continue working is available on the client when it goes offline.
- Design a store-and-forward mechanism where messages are created, stored while disconnected, and eventually forwarded to their respective destinations when a connection becomes available. The most common implementation of store-and-forward is a message queue.
- Determine how to deal with stale data, and how to prevent your Rich Client from using stale data.

Deployment Considerations

There are several options for deployment of Rich Client applications. You may have a stand-alone application where all of the application logic, including data, is deployed on the client machine. Another option is client/server, where the application logic is deployed on the client and the data is deployed on a database tier. Finally, there are several *n*-tier options where an application server contains part of the application logic.

Stand-Alone

Figure 2 illustrates a stand-alone deployment where all of the application logic and data is deployed on the client.

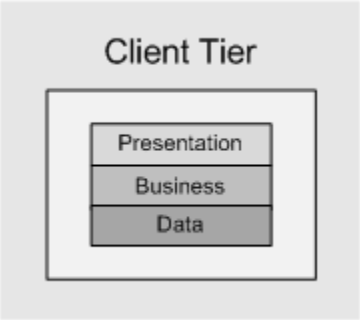

Figure 2 - Stand-alone deployment for a Rich Client application.

Client/Server

In a client/server deployment, all of the application logic is deployed on the client and the data is deployed on a database server, as shown in Figure 3.

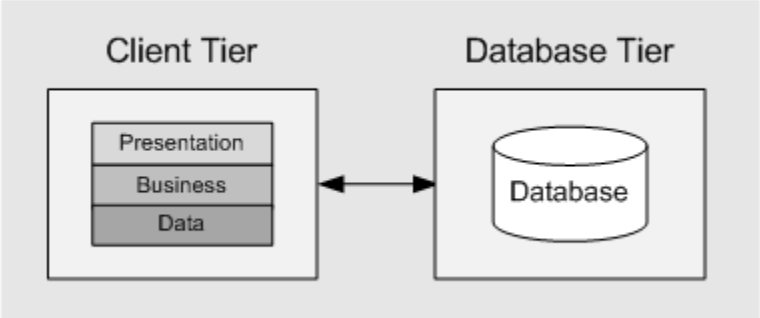

Figure 3 - Client/server deployment for a Rich Client application.

N-Tier

In an *n*-tier deployment, you can place presentation and business logic on the client, or just the presentation logic on the client. Figure 4 illustrates the case where the presentation and business logic are deployed on the client.

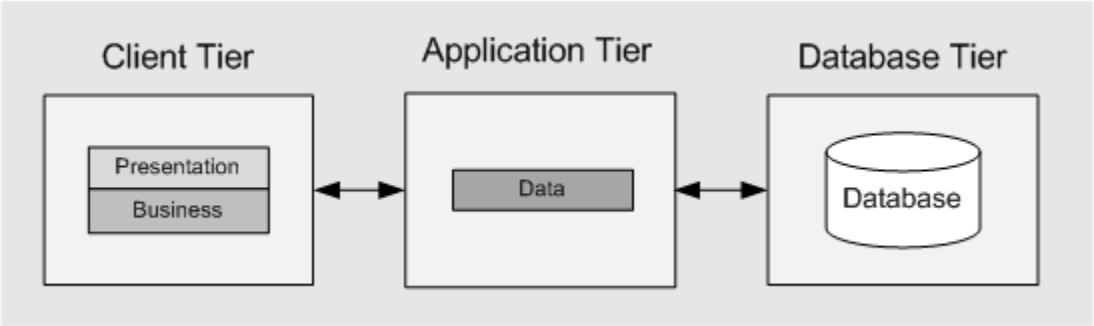

Figure 4 - N-tier deployment with the business layer located on the client tier.

Figure 5 illustrates the case where the business and data access logic are deployed on an application server.

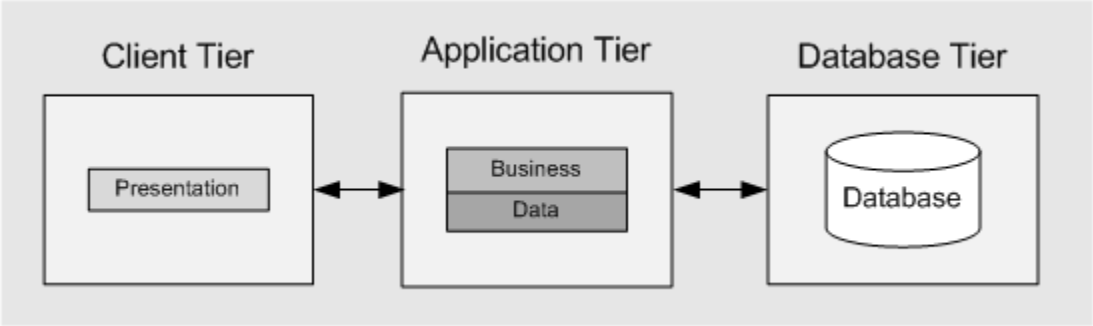

Figure 5 - N-tier deployment with the business layer located on the application tier.

Deploying Rich Client Applications

Several options are available for deploying a Rich Client application to a physical machine. Each has specific advantages and liabilities, and you should research the options to ensure that the one you choose is suitable for the target environments in which your application will execute.

The options are the following:

- No-touch deployment. The application executable is downloaded to the assembly download cache on the user's machine and executes in an environment that has constrained security settings.
- No-touch deployment with an application update stub. The application is automatically updated when the server-based version changes.
- XCOPY deployment. If no registry settings or component registration is required, the executable can be copied directly to the client machine hard disk.
- Windows Installer (.MSI) package. A comprehensive setup program that can install components, resources, registry settings, and other artifacts required by the application.
- XBAP (XML Browser Application) package. The application is downloaded through the browser and runs in a constrained security environment on the machine. Updates can be pushed to the client automatically.

Deployment Guidelines

It is important to know the deployment approach you will use for the application during the design phase, as this can limit the capabilities for deploying and installing artifacts that make up the application.

Consider the following deployment guidelines when you design a Rich Client application:

- Know your target physical deployment environment early, from the planning stage of the lifecycle.
- Clearly communicate the environmental constraints that drive software design and architecture decisions.

- Clearly communicate the software design decisions that require certain infrastructure attributes.

Pattern Map

Category	Relevant Patterns
Communication	• Asynchronous Callback • Gateway / Service Gateway • Service Locator • Service Agent and Proxy • Service Interface
Composition	• Composite View • Template View • Two-Step View • View Helper
Configuration Management	• Provider
Data Services	• Domain Model • Entity Translator • Data Transfer Object
Exception Management	• Exception Shielding
State Management	• Context Object
Workflow	• View Flow • Work Flow

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.
- **Composite View -** Combine individual views into a composite view
- **Context Object –** An object used to manage the current execution context.
- **Data Transfer Object –** 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 to 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.
- **Gateway –** Provide access to an external system through a common abstract interface so that consumers are not required to understand the external system interface.
- **Provider -** Implement a component that exposes an API that is different from the client API to allow any custom implementation to be seamlessly plugged in.

- **Service Interface -** A programmatic interface that other systems can use to interact with the service**.**
- **Service Locator -** Centralize distributed service object lookups, provide a centralized point of control, and act as a cache that eliminates redundant lookups.
- **Template View** - Implement a common template view, and derive or construct views using the template view.
- **Two-Step View -** Transform the model data into a logical presentation without any specific formatting, and then convert that logical presentation into the actual formatting required.
- **View Flow -** Manage navigation from one view to another based on state in the application or environment, and the conditions and limitations required for correct operation of the application.
- **View Helper –** Delegate business data processing responsibilities to helper classes.
- **Work Flow -** Manage the flow of control in a complex process-oriented application in a pre-defined manner while allowing dynamic route modification through decision and branching structures that can modify the routing of requests.

Technology Considerations

There are several different technologies available that you can use to implement a rich client application. The following guidelines will help you to choose an appropriate implementation technology, and provide guidance on the use of appropriate patterns and system functions for configuration and monitoring:

- If you want to build applications with good performance, interactivity, and have design support in Visual Studio, consider using Windows Forms.
- If you want to build applications that fully support rich media and graphics, consider using WPF.
- If you want to build applications that are downloaded from a Web server and then execute on the client, consider using XAML Browser Applications (XBAP).
- If you want to build applications that are predominantly document-based, or are used for reporting, consider designing an Office Business Application.
- If you decide to use Windows Forms and you are designing composite interfaces, consider using the Smart Client Software Factory.
- If you decide to use WPF and you are designing composite interfaces, consider using the Composite Application Guidance for WPF.
- If you decide to use WPF, consider using WPF Commands to communicate between your View and your Presenter or ViewModel.
- If you decide to use WPF, consider implementing the Presentation Model pattern by using DataTemplates over User Controls to give designers more control.
- If you want to support remote administration of configuration, or you need to obtain Windows Certification, consider implementing Group Policy overrides in your application.
- If you want to support remote monitoring for your application, consider using technologies such as SNMP and WMI to expose exceptions and health state.

Additional Resources

- For more information on designing Rich Client and Smart Client applications, see *Smart Client Architecture and Design Guide* at http://msdn.microsoft.com/en-us/library/ms998506.aspx.
- For more information on caching architectures, see *Caching Architecture Guide for .NET Framework Applications* at http://msdn.microsoft.com/en-us/library/ms978498.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.

Chapter 2 – Architecture and Design Guidelines

Objectives

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

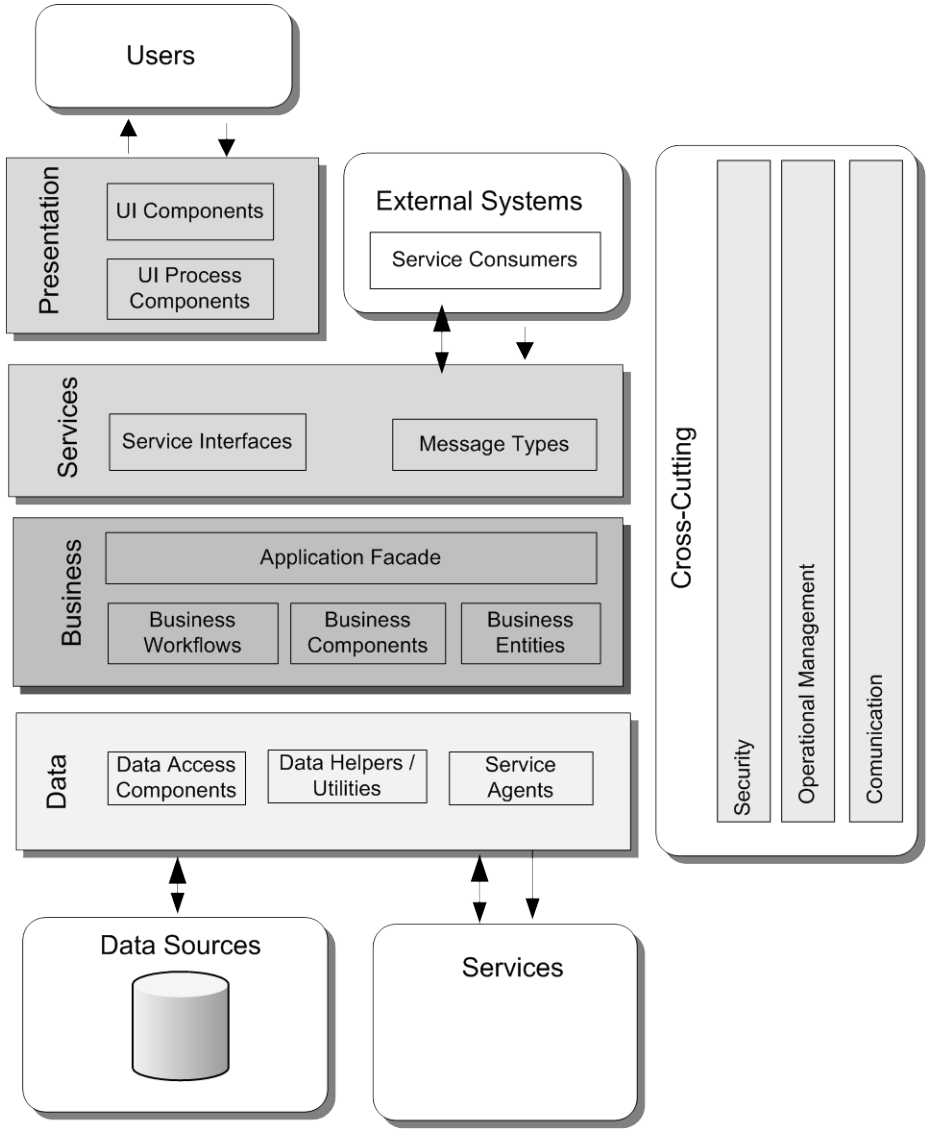

Figure 1. Common application architecture

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 an architecture that meets "best practice", minimizes costs and maintenance requirements, and promotes usability and extendibility. The key principles are the following:

- **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 you are not clear with requirements or if there are possibility of design evolution. This type of design often abbreviated as "BDUF".
- **Prefer Composition over Inheritance.** For reusing the functionality prefer using composition over inheritance, wherever possible, as inheritance increases dependency between parent and child classes 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 things 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. In other words, UI processing components should not include code that directly accesses a data source. Instead UI processing components should use either business components or data access components to retrieve data.

The following guidelines should be followed when designing an application:

- **Avoid 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, rather 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 is that a feature or functionality can be optimized independently of other features or functionality. Also if one feature fails it won't cause other features to fail as well, and they can run independently. It also helps to make the application easier to understand, design and manage complex interdependent systems.
- **Each component or module should have single responsibility.** Each component or a module should be responsible for only a specific feature or functionality. This makes your components cohesive helping 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.** A component or an 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 to appropriate sub-components 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. The functionality should not be duplicated in any other

component. Duplication of functionality within application leads to difficulty to change, decrease in clarity and potential inconsistency.

- **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 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.** Within a logical layer, the design of components should be consistent for a particular type. For example, if you choose to use the Table Data Gateway pattern to create an object that acts as a gateway to a table in a data source for data access, you should not include another pattern like Query Object to define an object that represents a database query.
- **You should not mix different types of components in the same logical layer**. For example, the User Interface (UI) layer should not contain business processing components. Instead, the UI layer 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 translates 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 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.** For reusing the functionality prefer using composition over inheritance, wherever possible, as inheritance increases dependency between parent and child classes limiting the reuse of child classes. This also reduces the inheritance hierarchies which can become quite hard 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 move 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 implement the cross-cutting concerns
- **Be consistent in the naming conventions used**. You should check 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 there, and you should not use exceptions to implement business logic. The standards should also include policies for logging and instrumentation as related to exceptions.

Architecture Frame

The following table lists the key areas to consider as you develop your architecture. Use 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.

Area	Key Issues
Authentication and Authorization	• Lack of authentication across trust boundaries. • Lack of authorization across trust boundaries. • Granular or improper authorization.
Caching	• Caching data that is volatile. • 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.
Composition	• Cooperating application modules are coupled by dependencies making development, testing, and maintenance more difficult. • Dependency changes between modules forces code recompilation and module redeployment. • Dynamic UI layout and update difficult due to hardcoded dependencies. • Dynamic module loading difficult due to hardcoded dependencies.

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.
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	• Per user authentication and authorization when not required. • Chatty calls to the database. • Business logic mixed with data access code.
Exception Management	• Leaving the application in an unstable state. • Revealing sensitive information to the end user. • Using exceptions for application logic. • Not logging sufficient details about the exception.
Layering	• Incorrect grouping of components within a layer. • Not following layering and dependency rules. • Not considering the physical distribution of layers.
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 runtime. • 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.
Structure	• Choosing the incorrect structure for your scenario. • Creating an overly complex structure when not required. • Not considering deployment scenarios.
User Experience	• Not following published guidelines. • Not considering accessibility • Creating overloaded interfaces with un-related functionality.
Validation	• Lack of validation across trust boundaries. • Not validating for all appropriate aspects of parameters, such as “Range”, “Type” and “Format”. • 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. Failing to design and implement a good authentication strategy can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attack.

When designing an authentication strategy, consider following guidelines:

- Identify your trust boundaries; authenticate users and calls across trust boundaries. Consider that calls may need to be authenticated from the client as well as from the server (mutual authentication).
- If you have multiple systems within the application which use different user repositories, consider a single sign-on strategy.
- 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.

Authorization

Designing a good authorization strategy is important for the security and reliability of your application. Failing to design and implement a good authorization strategy can make your application vulnerable to information disclosure, data tampering, and elevation of privileges.

When designing an authorization strategy, consider following guidelines:

- Identify your trust boundaries; authorize users and callers across trust boundary.
- Protect resources by applying authorization to callers based on their identity, groups, or roles.
- Use role-based authorization for business decisions.
- Use resource-based authorization for system auditing.
- Use claims-based authorization when you need to support federated authorization based on a mixture of information such as identity, role, permissions, rights, and other factors.

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.

When designing caching, consider following guidelines:

- 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.
- If your application is deployed in Web Farm, avoid using local caches that needs to be synchronized, instead consider using a transactional resource manager such as SQL Server or a product that supports distributed caching.
- 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.

When designing communication mechanisms, consider the following guidelines:

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

Composition

Composition is the process used to define how interface components in a user interface are structured to provide a consistent look and feel for the application. One of the goals with user interface design is to provide a consistent interface in order to avoid confusing users as they navigate through your application. This can be accomplished by using templates, such as a master page in ASP.NET, or by implementing one of many common design patterns.

When designing for composition, consider the following guidelines:

- Avoid using dynamic layouts. They can be difficult to load and maintain.
- Be careful with dependencies between components. Use abstraction patterns when possible to avoid issues with maintainability.
- Consider creating templates with placeholders. For example, use the Template View pattern to compose dynamic web pages to ensure reuse and consistency.
- Consider composing views from reusable modular parts. For example, use the Composite View pattern to build a view from modular, atomic component parts.

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 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. In a pessimistic model where two people modify a file concurrently, only the first person will be able to apply their changes to the original file. The other person will not be allowed to apply an update to the original 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.

When designing concurrency and transactions, consider the following guidelines:

- If you have business critical operations, consider wrapping them in transactions.
- Use connection-based transactions when accessing a single data source.
- Use Transaction Scope (System.Transaction) to manage transactions that span multiple data sources.
- 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.

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. Threading in general requires careful consideration when it comes to manipulating data that is shared by multiple threads and applying locks to that data.

When designing for concurrency at the application code level, consider the following guidelines:

- Updates to shared data should be mutually exclusive, which is accomplished by applying locks or using thread synchronization. This will prevent two threads from attempting to update shared data at the same time.
- Locks should be scoped at a very fine grained level. In other words, you should implement the lock just prior to making a modification and then release it immediately.
- When modifying static fields you should check the value, apply the lock, and check the value again before making the update. It is possible for another thread to modify the value between the point that you check the field value and the point that you apply the lock.
- Locks should not be applied against a type definition or the current instance of a type. In other words, you should not use lock (typeof(MyObject)) or Monitor.Enter(typeof(MyObject)) and you should not use lock(this) or Monitor.Enter(this). Using these constructs can lead to deadlock issues that are difficult to locate. Instead, define a private static field within the type and apply locks against the private field. You can use a common object instance when locking access to multiple fields or you can lock a specific field.

- 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. Failing to do so can make your application vulnerable to a variety of attacks, and also leads to an administrative overhead for your application.

When designing configuration management, consider following guidelines:

- Use least-privileged process and service accounts.
- Categorize the configuration items into logical sections if your application has multiple tiers.
- If your server application runs in a farm, decide which parts of the configuration are shares and which parts are specific to the machine the application is running on. Then choose an appropriate configuration store for each section.
- 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 only logging represents high cohesion.

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

- 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 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 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.

When designing data access components, consider the following guidelines:

- Do not couple your application model to your database schema.
- Open connections as late as possible and release them as early as possible.
- 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. Failing 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 consider providing access points within your exception management system to support instrumentation and centralized monitoring that assists system administrators.

When designing an exception management strategy, consider following guidelines:

- Do not catch internal 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.

When designing layers, consider the following guidelines:

- Layers should represent a logical grouping of components. For example, use separate layers for user interface, 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.
- When designing the interface for each layer, consider physical boundaries. If communication crosses a physical boundary to interact with the layer, use message-based operations. If communication does not cross a physical boundary, use object-based operations.
- 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. Failing 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. This frequently provides an 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 using performance counters and events. System monitoring tools, or other access points, can provide administrators with information about the state, performance, and health of an application.

When designing a logging and instrumentation strategy, consider following guidelines:

- 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 they can be modified at runtime 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 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.

When designing a state management mechanism, consider following guidelines:

- 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.
- If you are building a web application and performance is your primary concern, use an in-process state store such as ASP.NET session state variables.
- If you are building a web application and you want your state to persist through ASP.NET restarts, use the ASP.NET session state service.
- If your application is deployed in Web Farm, avoid using local state management stores that needs to be synchronized, instead consider using a remote session state service or the SQL server state store.

Structure

Software architecture is often defined as being the structure or structures of an application. When defining these structures, the goal of a software architect is to minimize the complexity by separating items into areas of concern using different levels of abstraction. You start by examining the highest level of abstraction while identifying different areas of concern. As the design evolves, you dive deeper into the levels, expanding the areas of concern, until all of the structures have been defined.

When designing the application structure, consider the following guidelines:

- Identify common patterns used to represent application structure such as Client/Server and N-Tier.
- Understand security requirements for the environment in which your application will be deployed. For example, many security policies require physical separation of presentation logic from business logic across different sub-nets.
- Consider scalability and reliability requirements for the application.
- Consider deployment scenarios for the application.

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.

When designing for an effective user experience, consider the following guidelines:

- Design for a consistent navigation experience. Use composite patterns for the look-and-feel, and controller patterns such as MVC, Supervising Controller, and Passive View, 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.

- Design similar components to have consistent behavior across the application. For example, a grid used to display data should implement a consistent interface for paging and sorting the data.
- Consider using published user interface guidelines. In many cases, an organization will have published guidelines that you should adhere to.

Validation

Designing an effective validation mechanism is important for the security and reliability of your application. Failing to do so can make your application vulnerable to cross-site scripting, SQL injection, buffer overflow, and other types of malicious input attack. 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 exploit of the vulnerability.

When designing a validation mechanism, consider following guidelines:

- Identify your trust boundaries, and validate all inputs across trust boundary.
- Centralize your validation approach, if it can be reused.
- Constrain, reject, and sanitize user input. In other words, assume all user input is malicious.
- Validate input data for length, format, and type.
- Do not rely only on client-side validation. Client-side validation will improve the user experience, but can be disabled. Server-side validation provides an additional layer of security.

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 process 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.

When designing a workflow component, consider the following guidelines:

- Determine management requirements. 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 un-deterministic nature of users. In other words, you cannot determine when a task will be completed, or if it will be completed correctly.

Pattern Map

Category	Relevant Patterns
Caching	• Cache Dependency

	• Page Cache
Communication	• Intercepting Filter • Pipes and Filters • Service Interface
Concurrency and Transactions	• Capture Transaction Details • Optimistic Offline Lock • Pessimistic Offline Lock
Coupling and Cohesion	• Adapter • Dependency Injection
Data Access	• Active Record • Data Mapper • Query Object • Repository • Row Data Gateway • Table Data Gateway
Layering	• Façade • Layered Architecture

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.
- **Capture Transaction Details** – Create database objects, such as triggers and shadow tables, to record changes to all tables belonging to the transaction.
- **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.
- **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 reduce coupling between systems.
- **Intercepting Filter** - A chain of composable filters (independent modules) that implement common pre-processing and post-processing tasks during a Web page request.
- **Optimistic Offline Lock** – Ensure that changes made by one session do not conflict with changes made by another session.
- **Page Cache** – Improve the response time for dynamic Web pages that are accessed frequently, but change less often and consume a large amount of system resources to construct.

- **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.
- **Repository** – An in-memory representation of a data source that works with domain entities.
- **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 authentication, see *Designing Application-Managed Authorization* at http://msdn.microsoft.com/en-us/library/ms954586.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, see *Designing Data Tier Components and Passing Data Through Tiers* at http://msdn.microsoft.com/en-us/library/ms978496.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 to design the presentation layer.
- Learn the key patterns and technology considerations.

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.

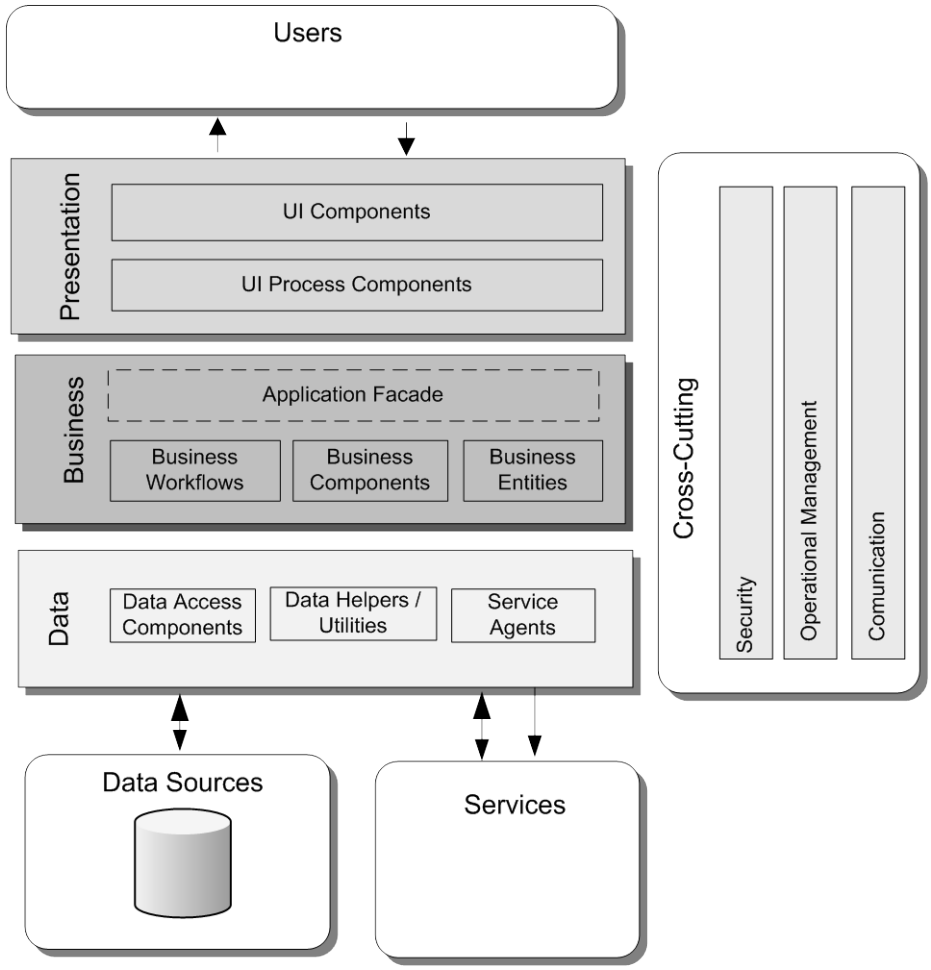

Figure 1 - A typical application showing the presentation layer and the components it may contain.

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 user interface. Implementing common user interaction patterns as separate user process components allows you to reuse them in multiple user interfaces.

Approach

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

1. Determine how you will present data. Choose the data format for your presentation layer and decide how you will present the data in your User Interface (UI).
2. Determine your data validation strategy. Use data validation techniques to protect your system from un-trusted input.
3. Determine your business logic strategy. Factor out your business logic to decouple it from your presentation layer code.
4. 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 Web presentation layer. Use the following principles to ensure that your design meets the requirements for your application, and follows best practices:

- **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 presentation entities to manage the data required to present your views. Use dedicated UI process components to manage the processing of user interaction.
- **Consider human interface guidelines.** Review your organization's guidelines for user interface design. Review established user interface guidelines based upon 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. • Failing to choose a suitable caching mechanism for use in a Web farm. • Assuming that data will still be available in the cache – it may have expired and been removed.
Composition	• Failing to consider use of patterns and libraries that support dynamic layout and injection of views and presentation at runtime. • Using presentation components that have dependencies on support classes and services instead of considering patterns that support run-time dependency injection. • Failing to use the Publish/Subscribe pattern to support events between components. • Failing to properly decouple the application as separate modules that can be added easily.
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 implement application logic. • Catching exceptions you do not handle. • Using custom exceptions when not necessary.
Input	• Failing to design for intuitive use, or implementing over-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 mobile devices, touch-screen, and pen and ink enabled devices.
Layout	• Using an inappropriate layout style for Web pages. • Implementing an overly-complex layout. • Failing to choose appropriate layout components and technologies. • Failing to adhere to accessibility and usability guidelines and standards. • Implementing an inappropriate workflow interface. • Failing to support localization and globalization.

Navigation	• Inconsistent navigation. • Duplication of logic to handle navigation events. • Using hard-coded navigation. • Failing to manage state with wizard navigation.
Presentation Entities	• Defining entities that are not necessary. • Failing to implement serialization when necessary.
Request Processing	• Blocking the user interface during long-running requests. • Mixing processing and rendering logic. • Choosing an inappropriate request-handling pattern.
User Experience	• Displaying unhelpful error messages. • Lack of responsiveness. • Over-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 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. • Relying only on client-side input validation. You must always validate input on the server or in the business layer as well. • 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.

When designing your caching strategy, consider the following guidelines:

- 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.
- If your application is deployed in Web Farm, avoid using local caches that needs to be synchronized, instead consider using a transactional resource manager such as SQL Server or a product that supports distributed caching.

- Do not depend on data still being in your cache. It may have been removed.

Composition

Consider whether your application will be easier to develop and maintain if the presentation layer uses independent modules and views that are easily composed at runtime. Composition patterns support the creation of views and the presentation layout at runtime. These patterns also help to minimize code and library dependencies that would otherwise force recompilation and redeployment of a module when the dependencies change. Composition patterns help you to implement sharing, reuse, and replacement of presentation logic and views.

When designing your composition strategy, consider the following guidelines:

- Avoid using dynamic layouts. They can be difficult to load and maintain.
- Be careful with dependencies between components. Use abstraction patterns when possible to avoid issues with maintainability.
- Consider creating templates with placeholders. For example use the Template View pattern to compose dynamic web pages to ensure reuse and consistency.
- Consider composing views from reusable modular parts. For example use the Composite View pattern to build a view from modular, atomic component parts.
- If you need to allow communication between presentation components, consider implementing the Publish/Subscribe pattern. This will lower the coupling between the components and improve testability.

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.

When designing your exception management strategy, consider the following guidelines:

- 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 case of business errors, display a user-friendly error message and allow user to retry the operation. In case of system exceptions, check if it is caused because of issues like system or database failure, display 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 upon 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.

When designing your input collection strategy, consider the following guidelines:

- Use forms-based input controls for normal data collection tasks.
- Use a document-based input mechanism for collecting input in Office-style documents.
- Implement a wizard-based approach for more complex data collection tasks, or 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 while 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 processing 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.

When designing your layout strategy, consider the following guidelines:

- Use templates to provide a common look and feel to all 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 Cascading Style Sheets (CSS) for layout whenever possible. This will improve rendering performance and maintainability.
- Use design patterns, such as Model-View-Presenter, 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.

When designing your navigation strategy, consider the following guidelines:
- Use well-known design patterns to decouple the user interface from the navigation logic where this logic is complex
- Design tool-bars and menus to help users find functionality provided by the UI.
- 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.

Presentation Entities

Use presentation entities to store the data you will use in your presentation layer to manage your views. Presentation entities are not always necessary; use them only if your data sets are sufficiently large and complex to require separate storage from the UI controls.

When designing presentation entities, consider the following guidelines:
- Determine if you require presentation entities. Typically, you may require presentations entities only if the data or the format to be displayed is specific to the presentation layer.
- If you are working with data-bound controls, consider using custom objects, collections, or DataSets as your presentation entity format.
- If you want to map data directly to business entities, use a custom class for your presentation entities.
- Do not add business logic to presentation entities.
- If you need to perform data type validation, consider adding it in your presentation entities.

Request Processing

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

When designing request processing, consider the following guidelines:
- Use asynchronous operations or worker threads to avoid blocking the user interface for long-running actions.
- Avoid mixing your user interface processing and rendering logic.
- Consider using the Passive View pattern (MVP) for interfaces that do not manage a lot of data.
- Consider using the Supervising Controller pattern (MVP) for interfaces that manage large amounts of data.

User Experience

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

When designing for user experience, consider the following guidelines:

- Utilize AJAX to improve responsiveness, and reduce postbacks and page reloads.
- Do not design overloaded or over-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 they interact 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.

When designing UI components, consider the following guidelines:

- Take advantage of the data-binding features of the controls you use in the user interface.
- 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 the control from scratch.
- Implement designer support for custom controls.
- Consider maintaining the state of controls as the user interacts with the application instead of reloading controls with each action.

UI Processing 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.

When designing UI processing components, consider the following guidelines:

- Don’t 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 runtime 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.

When designing your input and data validation strategy, consider the following guidelines:

- Validate all input data client-side where possible to improve interactivity and reduce errors caused by invalid data.
- Do not rely on just client side validation. Use server-side validation as well to constrain input for security purposes and to make security-related decisions.
- Design your validation strategy to constrain, reject, and sanitize malicious input.
- Use the built-in validation controls where possible.
- Consider using AJAX to provide real-time validation.

Pattern Map

Category	Relevant Patterns
Caching	• Cache Dependency • Page Cache
Composition	• Composite View • Transform View • Two-step View
Exception Management	• Exception Shielding
Layout	• Template View
Navigation	• Front Controller • Page Controller
Presentation Entities	• Entity Translator
User Experience	• Asynchronous Callback • Chain of Responsibility
UI Processing Components	• Model View Controller (MVC) • Passive View • Supervisor Controller

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.

- **Composite View** – Combine individual views into a composite representation.
- **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.
- **Front Controller** – Consolidate request handling by channeling all requests through a single handler object, which can be modified at runtime with decorators.
- **Model View Controller** – Separate the user interface 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 Cache** – Improve the response time for dynamic Web pages that are accessed frequently, but change less often and consume a large amount of system resources to construct.
- **Page Controller** – Accept input from the request and handle it for a specific page or action on a Web site.
- **Passive View** – Reduce the view to the absolute minimum by allowing the controller to process user input and maintain the responsibility for updating the view.
- **Presentation Model** – Move all view logic and state out of the view, and render the view through data-binding and templates.
- **Supervising Controller** – A variation of the MVC pattern in which the controller handles complex logic, in particular coordinating between views, but the view is responsible for simple view-specific logic.
- **Template View** – Implement a common template view, and derive or construct views using this template view.
- **Transform View** – Transform the data passed to the presentation tier into HTML for display in the UI.
- **Two-Step View** – Transform the model data into a logical presentation without any specific formatting, and then convert that logical presentation to add the actual formatting required.

Additional Resources

For more information on patterns, standards, and usability guidelines, see the following resources:

- *Microsoft Inductive User Interface Guidelines* at http://msdn.microsoft.com/en-us/library/ms997506.aspx.
- *User Interface Control Guidelines* at http://msdn.microsoft.com/en-us/library/bb158625.aspx.
- *User Interface Text Guidelines* at http://msdn.microsoft.com/en-us/library/bb158574.aspx.
- *Design and Implementation Guidelines for Web Clients* at http://msdn.microsoft.com/en-us/library/ms978631.aspx.
- *Web Presentation Patterns* at http://msdn.microsoft.com/en-us/library/ms998516.aspx.

Chapter 4 – Business Layers Guidelines

Objectives

- Understand how the business layer fits into the application architecture.
- Understand the components of the business layer.
- Learn the steps for designing these components.
- Learn the common issues faced while designing the business layer.
- Learn the key guidelines to design 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 represent the key areas for business layer design where mistakes occur most often. Figure 1. shows how the business layer fits into common application architecture.

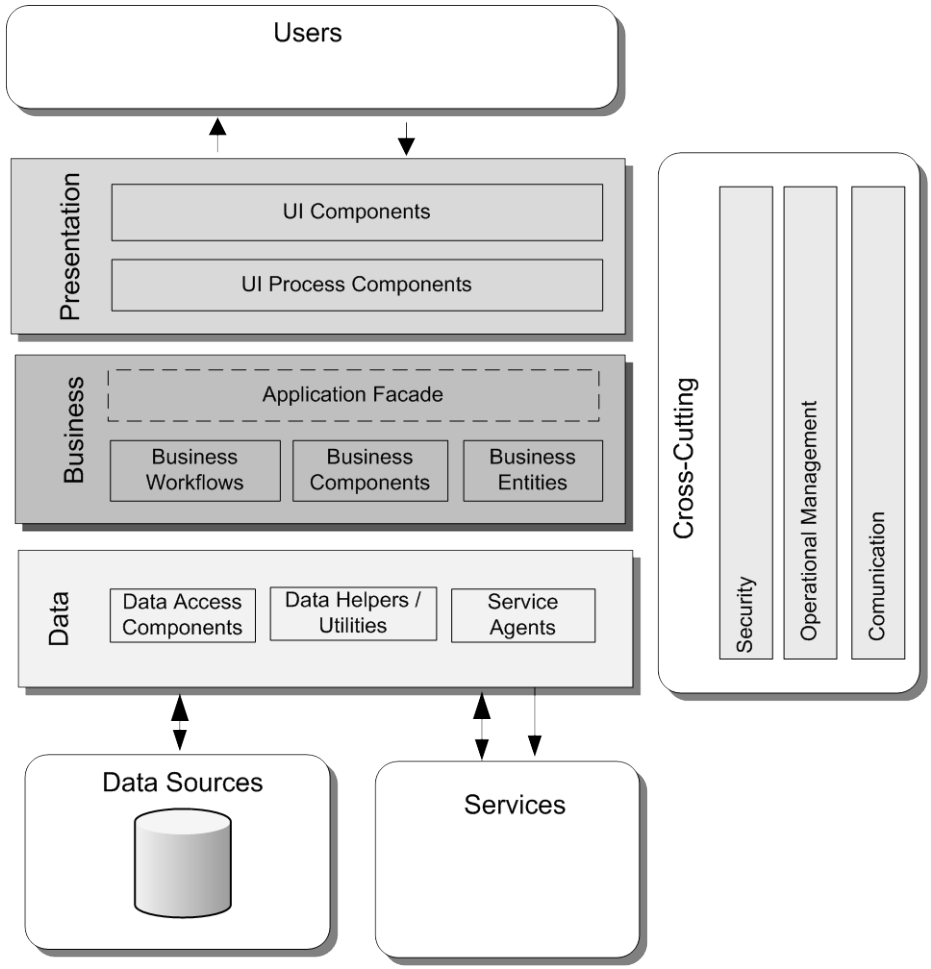

Figure 1 - A typical application showing the business layer and the components it may contain.

Business Components

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

- **Application Facade**. (Optional). An application façade combines multiple business operations into single message-based operation. You might access the application façade from the presentation layer using different communication technologies.
- **Business components**. After a user process collects the data it requires, the data can be operated on using business rules. The rules will describe how the data should be manipulated and transformed as dictated by the business itself. The rules may be simple or complex, depending on the business itself. The rules can be updated as the business requirements evolve.
- **Business entity components**. Business entities are used to pass data between components. The data represents real-world business entities, such as products and orders. The business entities that the application uses internally are usually data structures such as DataSets, Extensible Markup Language (XML) streams. Alternatively, they can be implemented using

custom object-oriented classes that represent the real-world entities your application has to work with, such as a product or an order.

- **Business workflow**. 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 you 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 what serialization support you will need.
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.

When designing the business layer, consider following guidelines:

- **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.
- **Identify the consumers of your business layer.** This will help to determine how you expose you business layer. For example, if your business layer will be used by your presentation layer and by an external application, you may choose to expose your business layer through a service.
- **Reduce round trips when accessing a remote business layer**. If you are using a message-based interface, consider using coarse-grained packages for data, such as Data Transfer Objects. In addition, consider implementing a remote façade for the business layer interface.
- **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. For Web applications, consider a message-based interface between the presentation layer and the business layer.

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
Authentication	• Applying authentication in a business layer when not required. • Designing a custom authentication mechanism. • Failing to use single-sign-on where appropriate.
Authorization	• Using incorrect granularity for roles. • Using impersonation and delegation when not required. • Mixing authorization code and business processing code.
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	• Using the Domain Model when not appropriate. • Choosing incorrect data formats for your business entities. • Not considering serialization requirements.
Caching	• Caching volatile data. • Caching too much data in the business layer. • Failing to cache data in a ready-to-use format. • Caching sensitive data in unencrypted form.
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 business layer. • Mixing data access logic within business logic in business components.
Exception Management	• Revealing sensitive information to the end user. • Using exceptions for application logic. • Not logging sufficient detail from exceptions.
Logging and Instrumentation	• Failing to add adequate instrumentation to business components. • Failing to log system-critical and business-critical events. • Not suppressing logging failures.
Service Interface	• Breaking the service interface. • Implementing business rules in the service interface. • Failing to consider interoperability requirements.
Validation	• Relying on validation that occurs in the presentation layer. • Not validating all aspects of parameters, such as “Range”, “Type” and “Format”. • 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.

Authentication

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

When designing an authentication strategy, consider following guidelines:

- Only authenticate users in the business layer if it is shared by other applications. If the business layer will be used only by a presentation layer or a service layer on the same tier, avoid authentication in the business layer.
- If your business layer will be used in multiple applications, using separate user stores, consider implementing a single-sign-on mechanism.
- Only flow the caller's identity to the business layer if you need to authenticate based on the original caller's ID.
- Consider using a trusted subsystem for access to back-end services to maximize the use of pooled database connections.
- If the presentation and business layers are deployed to the same machine and you need to access resources based on the original caller's ACL permissions, consider using impersonation.
- If the presentation and business layers are deployed to separate machines and you need to access resources based on the original caller's ACL permissions, consider using delegation. Only use delegation if it's absolutely necessary as many environments don't allow delegation. Instead authenticate the user at the boundary and use trusted subsystems in subsequent calls to lower layers.
- If using Web services, consider using IP Filtering to restrict call only from the presentation layer.

Authorization

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

When designing an authorization strategy, consider following guidelines:

- Protect resources by applying authorization to callers based on their identity, account groups, or roles.
- Use role-based authorization for business decisions.
- Use resource-based authorization for system auditing.
- Use claims-based authorization when you need to support federated authorization based on a mixture of information such as identity, role, permissions, rights, and other factors.
- Avoid using impersonation and delegation as it can significantly affect performance and scaling. It is generally more expensive to impersonate a client on a call than to make the call directly.

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.

When designing business components, consider following guidelines:

- 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.

When designing business entities, consider following guidelines:

- Choose appropriate data formats for your business entities. As a general rule, you should use custom objects. However, for smaller data-driven applications or document centric data, consider using XML for the data format.
- Consider analysis requirements and complexity associated with a Domain Model design before choosing to use it for business entities. A Domain Model is very good for handling complex business rules and works best with a stateful application.
- 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 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.

When designing a caching strategy, consider following guidelines:

- Cache 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.
- Consider how Web farm deployment will affect the design of your business layer caching solution. If a request can be handled by any server in the farm you will need to support the synchronization of cached data that can change.

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.

When designing for coupling and cohesion, consider following guidelines:

- 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.

When designing for concurrency and transactions, consider the following guidelines:

- Use connection-based transactions when accessing a single data source.
- 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 should an operation within the transaction fail.

- 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 to maximize 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.

When designing a data access strategy, consider the following guidelines:

- 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, and 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 internal 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.

When designing a logging and instrumentation strategy, consider following guidelines:

- 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.

Service Interface

When the business layer is deployed to a separate tier, or when implementing the business layer for a service, you must consider the guidelines for service interfaces. When designing a service interface, you must to consider the granularity of service operations and interoperability requirements. Generally, services should provide coarse-grained operations that reduce round-trips between the service and service consumer. In addition, you should use common data formats for the interface schema that can be extended without affecting consumers of the service.

When designing a service interface, consider following guidelines:

- Design your services 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 or in the service implementation layer.
- Design service interfaces for maximum interoperability with other platforms and services by using common protocols and data formats.
- Design the service to expose schema and contract information only, and make no assumptions on how the service will be used.
- Choose an appropriate transport protocol. For example, choose named pipes or shared memory when the service and service consumer are on the same physical machine, TCP when a service is accessed by consumers within the same network, or HTTP for services exposed over the Internet.

Validation

Designing an effective validation 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 cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attack. 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.

When designing a validation strategy, consider following guidelines:

- 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 all user input is malicious.
- Validate input data for length, 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 understand the options that are available.

When designing a workflow strategy, consider the following guidelines:

- 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.

Deployment Considerations

When deploying a business layer, you must consider performance and security issues within the production environment.

When deploying a business layer, consider following guidelines:

- Deploy the business layer to the same physical tier as the presentation or service layer to maximize application performance.
- If you must support a remote business layer, consider using TCP protocol to improve performance of the application.
- Use IPSec to protect data passed between physical tiers for all business layers for all applications.
- Use SSL to protect calls from business layer components to remote Web services.

Pattern Map

Category	Relevant Patterns
Business Components	• Application Façade • Chain of Responsibility • Command

Business Entities	• Domain Model • 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 • Data Mapper • Query Object • Repository • Row Data Gateway • Table Data Gateway
Workflows	• Data-driven workflow • Human workflow • Sequential workflow • State-driven workflow

Pattern Descriptions

- **Active Record** – Include a data access object within a domain entity.
- **Application Façade** – Centralize and aggregate behavior to provide a uniform service layer.
- **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 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-driven Workflow** – A workflow that contains tasks whose sequence is determined by the values of data in the workflow or the system.
- **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 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.
- **Repository** – An in-memory representation of a data source that works with domain entities.
- **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 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).
- If you require workflows that implement complex orchestrations and support reliable store and forward messaging capabilities, consider using BizTalk Server.
- If you must interact with non-Microsoft systems, perform EDI operations, or implement Enterprise Service Bus (ESB) patterns, consider using the ESB Guidance for BizTalk Server.
- If your business layer is confined to a single SharePoint site and does not require access to information in other sites, consider using MOSS. MOSS is not suitable for multiple-site scenarios.
- If you are designing transactions that span multiple data sources, consider using a transaction scope (System.Transaction) to manage the entire transaction.

Additional Resources

For more information, see the following resources:

- *Concurrency Control* at http://msdn.microsoft.com/en-us/library/ms978457.aspx.
- *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 while designing the data layer.
- Learn the key guidelines to design the data layer.
- Learn the key patterns and technology considerations.

Overview

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

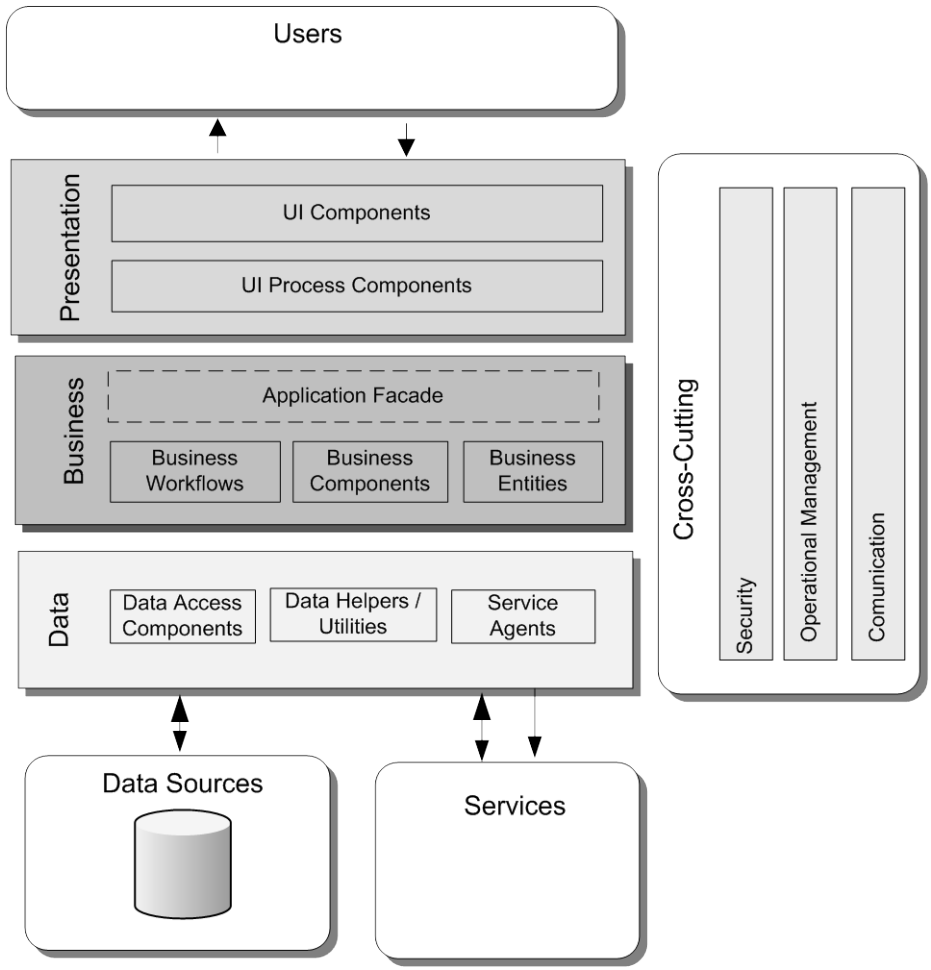

Figure 1 - A typical application showing the data layer and the components it may contain.

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 may 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. The following sections of this guide discuss these options and enumerate the benefits and drawbacks of 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 documents. 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 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 and scalability objectives**. Scalability and performance objectives for the data access layer should be taken into account during design. For example, when designing an Internet-based merchant application, data layer performance is likely to be a bottleneck for the application. When data layer performance is critical, use profiling to understand and then limit expensive data operations.

Data 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 database. • Searching and manipulating BLOB data.
Batching	• Failing to use batching to reduce database round-trips . • Holding onto locks for excessive periods when batching. • Failing to consider a strategy for reducing database round-trips with batching.
Connections	• Improper configuration of connection pooling. • Failing to handle connection timeouts and disconnections. • Performing transactions that span multiple connections. • Holding connections open for excessive periods. • Using individual identities instead of a trusted subsystem to access the database.
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.
Stored Procedures	• Not passing parameters to stored procedures correctly. • Implementing business logic in stored procedures. • Not considering how dynamic SQL in stored procedures can impact performance, security, and maintainability.
Transactions	• Using the incorrect isolation level. • Using exclusive locks, which can cause contention and deadlocks. • Allowing long-running transactions to blocking access to data.
Validation	• Failing to perform data type validation against 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. • Failure to set up proper indexes on applications that do heavy querying with XML • Failing to validate XML inputs using schemas.

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. BLOBs may have structure within them, 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

When designing for BLOBs, consider the following guidelines:

- Store images in a database only when it is not practical to store them on the disk.
- Use 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.
- Do not consider storing BLOB in the database when using buffered transmission.

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 is better because the database caches and can reuse a query execution plan for a similar query.

When designing batching, consider the following guidelines:

- Use 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
- Use batched commands and a DataReader to load or copy multiple sets of data.
- When loading large volumes of file-based data into the database, use 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

When designing for data layer connections, consider the following guidelines:

- In general, open connections as late as possible and close them as early as possible.
- To maximize the effectiveness of connection pooling, use a trusted sub-system security model and avoid impersonation if possible.
- Perform transactions through a single connection where possible.
- For security reasons, avoid using a System or User Data Source Name (DSN) to store connection information.
- 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 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 to allow the storage and retrieval of application state by the business layer.

When designing your data format, consider the following guidelines:

- 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. However, new O/RM solutions are available to reduce the amount of custom code required.

- Use 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 serilization 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.

When designing your exception management strategy, consider the following guidelines:

- 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.

Object Relational Mapping Considerations

When designing an Object Oriented (OO) application, consider the impedance mismatch between the OO model and the relational model that makes it difficult to translate between them. For example, encapsulation in OO designs, where fields are hidden, contradicts the public nature of properties in a database. Other examples of impedance mismatch include differences in the data types, structural differences, transactional differences, and differences in how data is manipulated. The two common approaches to handling the mismatch are data access design patterns such as Repository, and Object/Relational Mapping (O/RM) tools. A common model associated with OO design is the Domain Model, which is based on modeling entities after objects within a domain. As a result, the term domain represents an object-oriented design in the following guidelines.

When designing for object relational mapping, consider following guidelines:

- Consider using or developing a framework that provides a layer between domain entities and the database.
- If you are working in a Greenfield environment, where you have full control over the database schema, choose an O/RM tool that will generate a schema to support the object model and provide a mapping between the database and domain entities.

- If you are working in a Brownfield environment, where you must work with an existing database schema, consider tools that will help you to map between the domain model and relational model.
- If you are working with a smaller application or do not have access to O/RM tools, implement a common data access pattern such as Repository. With the Repository pattern, the repository objects allow you to treat domain entities as if they were located in memory.
- When working with Web applications or services, group entities and support options that will partially load domain entities with only the required data. This allows applications to handle the higher user load required to support stateless operations, and limit the use of resources by avoiding holding initialized domain models for each user in memory.

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:

- 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.
- Use objects to build the query. 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.

Stored Procedures

In the past, stored procedures represented a performance improvement over dynamic SQL statements. However, with modern database engines, performance is no longer a major factor. When considering the use of stored procedures, the primary factors are abstraction, maintainability, and your environment. This section contains guidelines to help you design your application when using stored procedures. For guidance on choosing between using stored procedures and dynamic SQL statements, see the section that follows.

When it comes to security and performance, the primary guidelines are to use typed parameters and avoid dynamic SQL within the stored procedure. Parameters are one of the factors that influence the use of cached query plans instead of rebuilding the query plan from scratch. When parameter types and the number of parameters change, new query execution plans are generated, which can reduce performance.

When designing stored procedures, consider the following guidelines:
- Use typed parameters as input values to the procedure and output parameters to return single values.
- Use parameter or database variables if it is necessary to generate dynamic SQL within a stored procedures.
- Consider using XML parameters for passing lists or tabular data.
- Design appropriate error handling and return errors that can be handled by the application code.
- Avoid the creation of temporary tables while processing data. However, if temporary tables need to be used, consider creating them in-memory rather than on disk.

Stored Procedures vs. Dynamic SQL

The choice between stored procedures and dynamic SQL focuses primarily on the use of SQL statements dynamically generated in code instead of SQL implemented within a stored procedure in the database. When choosing between stored procedures and dynamic SQL, you must consider the abstraction requirements, maintainability, and environment constraints.

The main advantages of stored procedures are:
- They provide an abstraction layer to the database, which can minimize the impact on application code when the database schema changes.
- Security is easier to implement and manage because you can restrict access to everything except the stored procedure.

The main advantages of dynamic SQL statements are:
- You can take advantage of fine-grained security features supported by most databases.
- They require less in terms of specialist skills than stored procedures.
- They are easier to debug than stored procedures.

When choosing between stored procedures and dynamic SQL. Consider the following guidelines:
- If you have a small application that has a single client and few business rules, dynamic SQL is often the best choice.
- If you have a larger application that has multiple clients, consider how you can achieve the required abstraction. Decide where that abstraction should exist: at the database in the form of stored procedures, or in the data layer of your application in the form of data access patterns or object/relational mapping (O/RM) products.
- If you want to minimize code changes when the database schema changes, consider using stored procedures to provide an abstraction layer. Changes associated with normalization or schema optimization will often have no affect on application code. If a schema change does affect inputs and outputs in a procedure then application code is affected; however, the changes are limited to clients of the stored procedure.

- Consider the resources you have for development of the application. If you do not have resources intimately familiar with database programming, consider tools or patterns that are more familiar to your development staff.
- Consider debugging support. Dynamic SQL is easier for application developers to debug.
- When considering dynamic SQL, you must understand the impact that changes to database schemas will have on your application. You must provide an abstraction in the data layer to decouple the interface between business components and the database when not using stored procedures.

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 made permanent. Transactions support undo (rollback) database actions following an error, which helps to preserve the integrity of data in the database.

When designing transactions, consider the following guidelines:

- Enable transactions only when you need them. For example, you should not use a transaction for an individual SQL statement because SQL Server 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.
- If using manual or explicit transactions, consider implementing the transaction within a stored procedure.
- Consider the use of Multiple Active Result Sets (MARS) in transaction heavy concurrent applications to avoid potential deadlock issues.

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, 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.

- 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

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.

When designing for the use of XML, consider the following guidelines:

- Use XML readers and writers to access XML-formatted data.
- Use an XML schema to define formats and to provide validation for data stored and transmitted as XML.
- Use custom validators for complex data parameters within your XML schema.
- Store XML in typed columns in the database, if available, for maximum performance.
- For read-heavy applications that use XML in SQL Server, consider XML indexes.

Manageability Considerations

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

When designing for manageability, consider the following guidelines:

- Use 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 SQL Server databases directly, all applications deployed to clients that use the business entities will require SQL connectivity and logon permissions.
- Use stored procedures to abstract data access from the underlying data schema. However, be careful not to overuse them because this will severely impact code maintenance and reuse and thus the maintainability of your application. A symptom of overuse is a large trees of stored procedures that call each other. Avoid using them to implement control flow, to manipulate individual values (for example, perform string manipulation), and other functionality difficult to implement in Transact-SQL.

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.

When designing for performance, consider the following guidelines:

- Use connection pooling and 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.
- Use 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 to 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. It should also protect the mechanisms used to gain access to the data source.

When designing for security, consider the following guidelines:

- When using Microsoft SQL Server, consider using Windows authentication with a trusted sub-system.
- Encrypt connection strings in configuration files instead of using a system or user Data Source Name (DSN).
- 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.

Deployment Considerations

When deploying a data access layer, the goal of a software architect is to consider the performance and security issues in the production environment.

When deploying the data access layer, consider the following guidelines:

- Locate the data access layer on the same tier as the business layer to improve application performance.

- If you need to support a remote data access layer, consider using the TCP protocol to improve performance.
- You should not locate the data access layer on the same server as the database.

Pattern Map

Category	Relevant Patterns
General	• Active Record • Application Service • Domain Model • Data Transfer Object • Repository • Table Data Gateway • Table Module
Batching	• Parallel Processing • Partitioning
Transactions	• Coarse Grained Lock • Capture Transaction Details • Implicit Lock • Optimistic Offline Lock • Pessimistic Offline Lock • Transaction Script

Pattern Descriptions

- **Active Record -** Include a data access object within a domain entity.
- **Application Service –** Centralize and aggregate behavior to provide a uniform service layer.
- **Capture Transaction Details –** Create database objects, such as triggers and shadow tables, to record changes to all tables belonging to the transaction.
- **Coarse Grained Lock –** Lock a set of related objects with a single lock.
- **Data Transfer Object** - 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.
- **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.

- **Repository -** An in-memory representation of a data source that works with domain entities.
- **Table Data Gateway -** An object that acts as a gateway to a table or view in a data source and centralizes all 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 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 require support for more complex data-access scenarios, or need to simplify your data access code, consider using the Enterprise Library Data Access Application Block.
- If you are building a data-driven Web application with pages based on the data model of the underlying database, consider using ASP.NET Dynamic Data.
- If you want to manipulate XML-formatted data, consider using the classes in the System.Xml namespace and its subsidiary namespaces.
- If you are using ASP.NET to create user interfaces, consider using a DataReader to access data to maximize rendering performance. DataReaders are ideal for read-only, forward-only operations in which each row is processed quickly.
- If you are accessing Microsoft SQL Server, consider using classes in the ADO.NET SqlClient namespace to maximize performance.
- If you are accessing Microsoft SQL Server 2008, consider using a FILESTREAM for greater flexibility in the storage and access of BLOB data.
- If you are designing an object oriented business layer based on the Domain Model pattern, consider using the ADO.NET Entity Framework.

patterns & practices Solution Assets

For information about p&p solution assets, see the following resources:

- Enterprise Library - *Data Access Application Block* at http://msdn.microsoft.com/en-us/library/cc309504.aspx
- *Performance Testing Guidance* at http://www.codeplex.com/PerfTesting/Wiki/View.aspx?title=Whats%20New&referringTitle=Home

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
- *Using stored procedures instead of SQL statements* at http://msdn.microsoft.com/en-us/library/ms978510.aspx.
- *.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 the application architecture.
- Understand the components of the service layer.
- Learn the steps for designing the service layer.
- Learn the common issues faced while designing the service layer.
- Learn the key guidelines to design the service layer.
- Learn the key patterns and technology considerations.

Overview

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 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 in the overall design of your application.

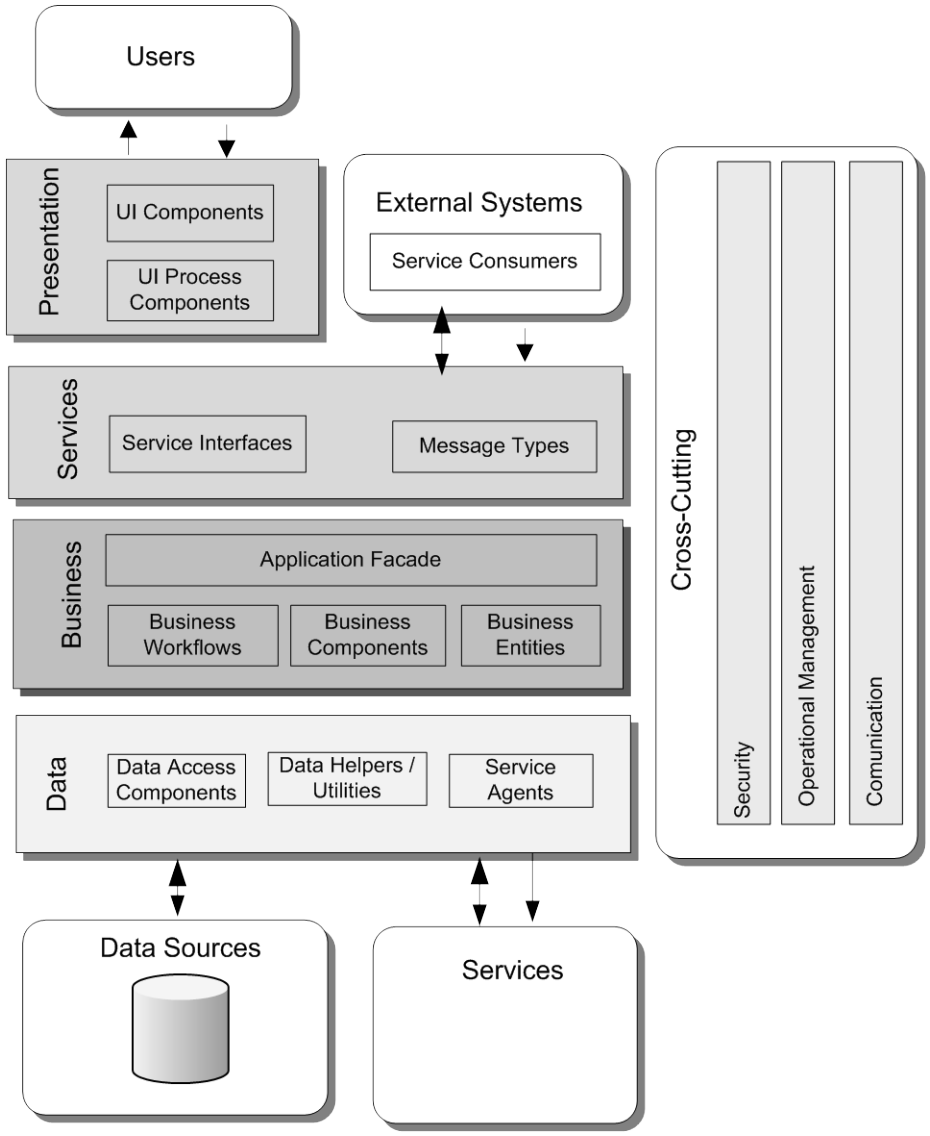

Figure 1 - An overall view of a typical application showing the service layer.

Service Layer 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 is defined, the next step is to design the service implementation; which is used to translate data contracts into business entities and 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

When designing the service layer, there are many factors that you should consider. Many of the 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.

- **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 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 without the assumption that you know who the client is**. You should not make assumptions about the client, or about how they plan to use the service that you provide.
- **Design only for 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 are 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 arrive out of order, implement a design that will store messages and then process them in the correct order.
- **Versioning of Contracts.** A new version for service contracts mean new operations exposed by the service whereas for data contracts it means new schema type definitions being added.

Services 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.

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.
Data Consistency	• Failing to check for data consistency. • Improper handling of transactions in a disconnected model.
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.
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	• There is limited schema support. • Current tools for REST are primitive. • 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. Failing to design a good authentication strategy can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attack.

When designing an authentication strategy, consider following guidelines:

- 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 SSL are used with basic authentication, or when credentials are passed as plain text.
- Use 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. Failing to design a good authorization strategy can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges.

When designing an authorization strategy, consider following guidelines:

- 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 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 TCP for more efficient communications. If the service will be deployed in to a public facing network, you should choose the HTTP protocol.

When designing a communication strategy, consider following guidelines:

- Determine how to handle unreliable or intermittent communication.
- Use 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.
- Decide if message communication must be one-way or two-way.

Data Consistency

Designing for data consistency is critical to the stability and integrity of your service implementation. Failing to validate the consistency of data received by the service can lead to invalid data being inserted into the data store, unexpected exceptions, and security breaches. As a result, you should always include data consistency checks when implementing a service.

When designing for data consistency, consider following guidelines:

- Validate all parameters passed to the service components.
- Check input for dangerous or malicious content.
- Determine your signing, encryption and encoding strategies.
- Use an XML schema to validate incoming SOAP messages.

Exception Management

Designing an effective exception management strategy for your service layer is important for the security and reliability of your application. Failing to do so can make your application vulnerable to denial of service (DoS) attacks, and may also allow it to reveal sensitive and critical information.
Raising and handling exceptions is an expensive operation, and it is important for the design to take into account the 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.

When designing an exception management strategy, consider following guidelines:

- 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 WCF. You must understand which patterns your chosen infrastructure supports, and determine the appropriate channel for interaction with consumers of the service.

When designing message channels, consider following guidelines:

- 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.

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 type 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.

When designing a message construction strategy, consider following guidelines:

- 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.

When designing message endpoints, consider following guidelines:

- 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 that 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 may 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 from tampering.

When designing message protection, consider following guidelines:
- If interactions between the service and the consumer are not routed through other services, you can use just transport layer security 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.
- Use 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 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.

When designing message routing, consider following guidelines:
- 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, the router must ensure 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.

When designing message transformation, consider following guidelines:

- 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.

Representational State Transfer (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 operations supported by a resource represent the functionality provided by a service that uses REST.

When designing REST resources, consider following guidelines:

- 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 GUID, for specific resource instances. For example, http://www.contoso.com/employee/8ce762d5-b421-6123-a041-5fbd07321bac4 represents an employee starting point while with a GUID that represents a specific employee appended to it.
- 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.

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 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.

When designing a service interface, consider following guidelines:

- Use a coarse-grained interface that minimizes the number of calls required to achieve a specific result.

- Design services 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.
- Use standard formats for parameters to provide maximum compatibility with different types of client.
- 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 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.

When designing SOAP messages, consider following guidelines:

- 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 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 to minimize performance impact when exposing business functionality.

When deploying the service layer, consider following guidelines:

- 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 located on the same physical tier as the service consumer, consider using named pipes or shared memory protocols.
- If the service is accessed only by other applications within a local network, consider using TCP for communications.
- If the service is publicly accessible from the Internet, use HTTP for your transport protocol.

Pattern Map

Category	Relevant Patterns
Communication	• Duplex • Fire and Forget • Reliable Sessions

	• Request Response
Data Consistency	• Atomic Transactions • Cross-service Transactions • Long running transactions
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
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

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.
- **Atomic Transactions -** Transactions that are scoped to a single service operation.
- **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 -** Use 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 can receive messages and invoke functionality inside the application.
- **Claim Check -** Retrieve data from a persistent store when required.
- **Command Message -** A message structure used to support commands.
- **Competing Consumer -** Set 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 -** A component that enriches messages with missing information obtained from an external data source.
- **Content Filter -** Remove sensitive data from a message and reduce network traffic by removing unnecessary data from a message.
- **Content-Based Router -** Route each message to the correct consumer based on the contents of the message; such as existence of fields, specified field values, and so on.
- **Cross-service Transactions -** Transactions that can span multiple services.
- **Data Confidentiality -** Use 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 -** Ensure that messages have not been tampered with in transit.
- **Data Origin Authentication -** Validate 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 application.
- **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 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 -** Prevent a service from exposing information about its internal implementation when an exception occurs.
- **Façade** – Implement a unified interface to a set of operations 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 -** Ensure that a service will only handle a message once.
- **Long-running Transaction -** Transactions that are part of a workflow process.
- **Message Broker (Hub-and-Spoke) -** A central component that communicates with multiple applications to receive messages from multiple sources, determine the correct destination, and route the message to the correct channel.
- **Message Bus -** Structure the connecting middleware between applications as a communication bus that enables them to work together using messaging.
- **Message Dispatcher -** A component that sends messages to multiple consumers.
- **Message Filter -** Eliminate 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 -** Encapsulate message-based calls into a single interface in order to separate it from the rest of the application code.
- **Messaging Mapper -** Transform requests into business objects for incoming messages, and reverse the process to convert business objects into response messages.

- **Normalizer -** Convert or transform data into a common interchange format when organizations use different formats.
- **Point-to-point Channel -** Send 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 -** Create 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 –** Create 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 -** Enforce 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 -** Use 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 -** Check 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 Windows Communication Foundation (WCF) services for advanced features and support for multiple transport protocols.
- If you are using ASP.NET Web Services, 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 and you want to support clients within an intranet, consider using the TCP protocol and binary message encoding with transport security and Windows authentication.
- If you are using WCF and you want to support WCF clients on the same machine, consider using the named pipes protocol and binary message encoding.
- 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 the following resources:

- *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn.microsoft.com/en-us/library/ms998469.aspx.
- *Web Service Security Guidance* at http://msdn.microsoft.com/en-us/library/aa480545.aspx
- *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.
- 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. 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. 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.

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 or named pipes protocols to package data into a single call that can be serialized across process boundaries.
- **Consider message-based communication when crossing physical boundaries.** 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.
- **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 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.

- 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:

- 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 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:

- Avoid fine-grained "chatty" interfaces for cross-process and cross-machine communication. 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.
- 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 with Web services using messages not based on SOAP for applications with formats such as RSS.
- 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-MetadataExchange in SOAP requests to obtain descriptive information about a service, such as its WSDL definition and policies.
- 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 WS-Coordination to coordinate two-phase commit transactions in the context of Web services conversations.
- 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:

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

- ASPX services support the endpoint technology implemented in IIS.
- ASPX 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 Web applications.
- 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 Web applications, including distributed and non-distributed styles, ways to scale the hardware, and the patterns that describe performance, reliability, and security issues. By considering the possible deployment scenarios for your application as part of the design process, you prevent a situation where the application cannot be successfully deployed, or fails to perform to its design requirements due to 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:

- Know your target physical deployment environment early when you are planning your design and architecture.
- Clearly understand and communicate the environmental constraints that drive software design and architecture decisions.
- Clearly communicate the software design decisions that drive specific infrastructure requirements.

Distributed vs. Non-Distributed Deployment

When creating your deployment strategy first determine if you will use a distributed or a non-distributed deployment model. If you are building a simple application for which you want to

minimize the number of required servers, consider a non-distributed deployment. If you are building a more complex application that you will want to optimize for scalability and maintainability, consider a distributed deployment.

Non-Distributed Deployment

A non-distributed deployment is where 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

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. Since all of your layers are sharing resources, one layer can negatively impact all the other layers when it is under heavy utilization. The use of a single tier reduces your overall scalability and maintainability because all of the layers share the same physical hardware.

Distributed Deployment

A distributed deployment is where 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.

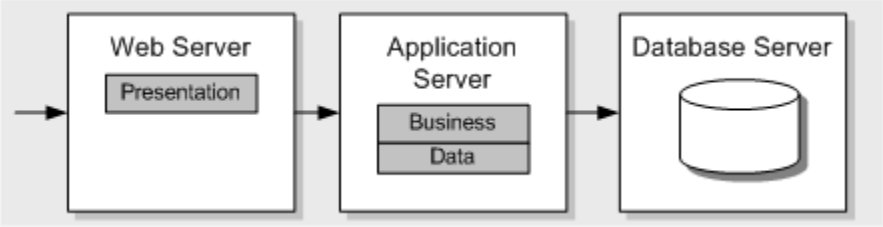

Figure 2. Distributed deployment

A distributed approach allows you to configure the application servers that host the various layers 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 applications servers and by using different authentication and authorization options. For example, in a rich client application, the client may use Web services

exposed through a Web server, or may access functionality in the application server tier using 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 due to 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.
- Use 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 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.
- 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 you are designing a distributed deployment, you will need to determine which layers and components you 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.
- Deploy business components that are used synchronously by user interfaces or user process components in the same physical tier as the user interface to maximize performance and ease operational management.
- Do not locate UI and business components on the same tier if there are security implications that require a trust boundary between them.
- 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 code that uses them.

Scale Up vs. Scale Out

Your approach to scaling is a critical design consideration because 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 to your existing servers to support increased capacity. This is a simple option and one that can be cost effective. It does not introduce additional maintenance and support costs. However, any single points 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, runtime, 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. It 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 level of performance. 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. To ensure that your application can be scaled out successfully, consider the following practices in your design:

- **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 SQL Server™ 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 may 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 may 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 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 due to 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 you are already running on. 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 loading. 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 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.

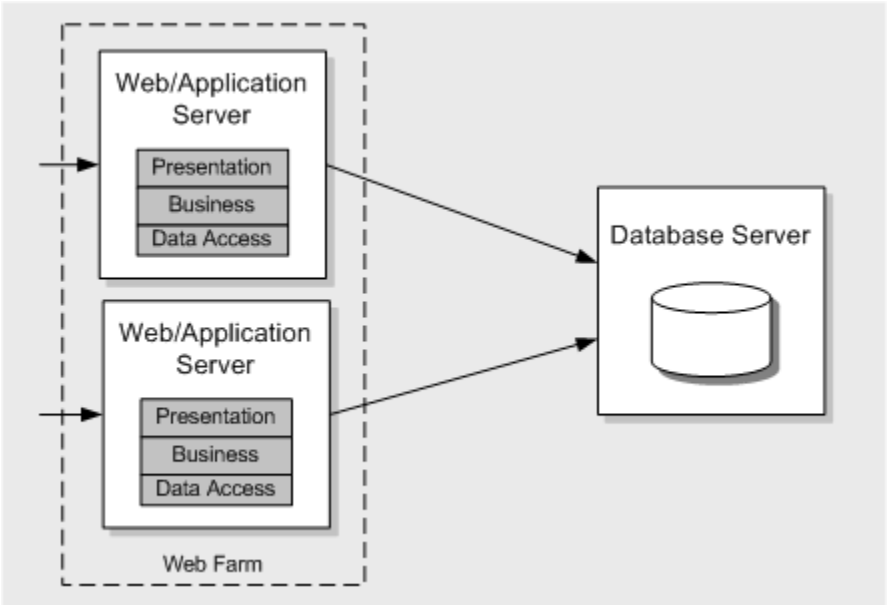

Figure 3. A simple Web farm

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 SSL, 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 using an application farm. An application farm is a collection of servers that run the same application. Requests from the presentation tier are distributed to each server in the farm so that each has approximately the same loading. 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 a load-balanced cluster.

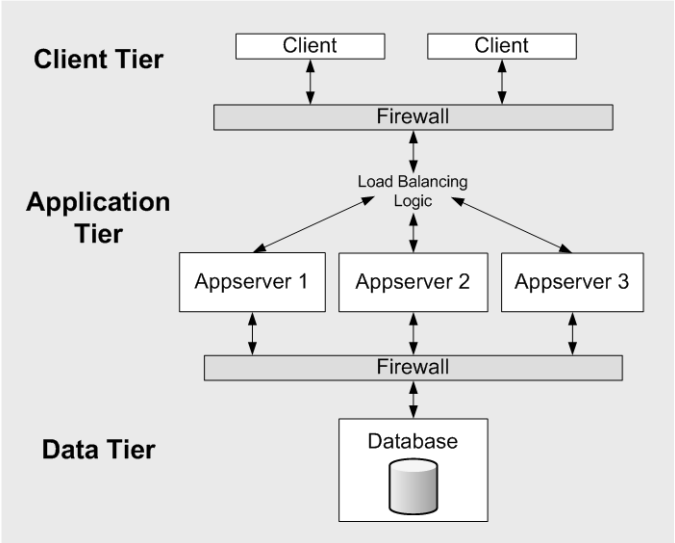

Figure 4. A load-balanced cluster

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 may 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 fail-over cluster to ensure the availability of your application even if a server fails.

Fail-Over Cluster

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

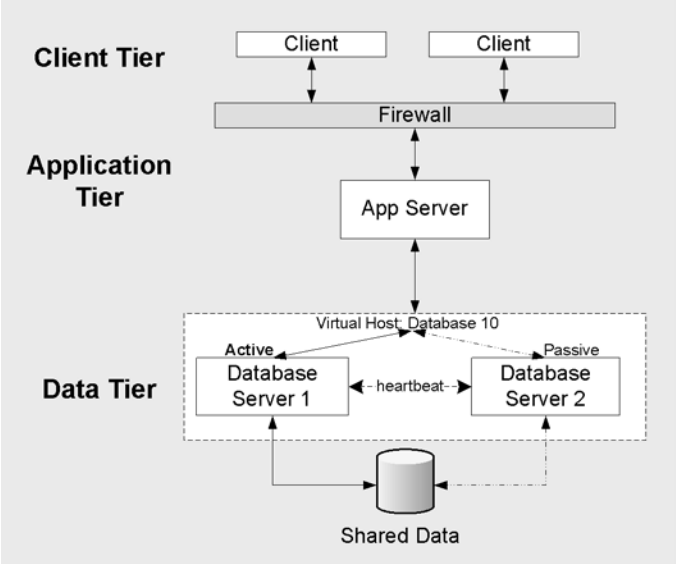

Figure 5. A failover cluster

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 and 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 operation (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.

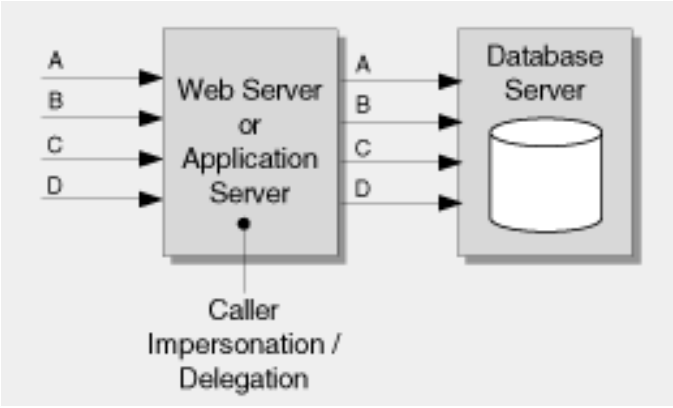

Figure 6. The impersonation/delegation authorization model

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.

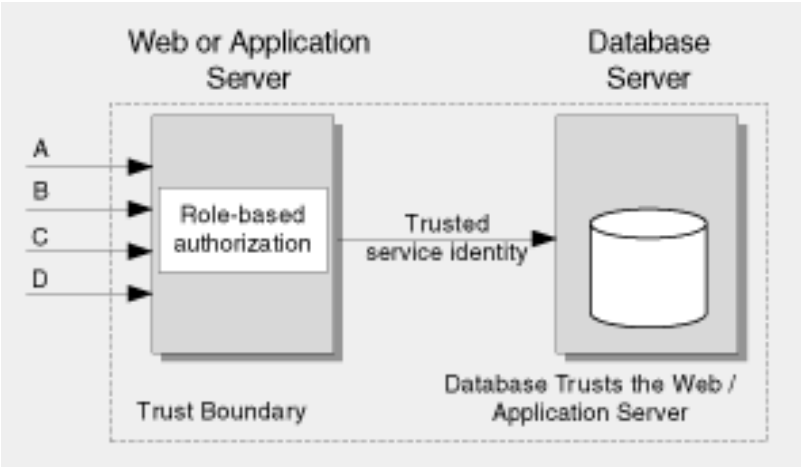

Figure 7. The trusted subsystem (or trusted server) model

Multiple Trusted Service Identities

In some situations, you may require more than one trusted identity. For example, you may 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.

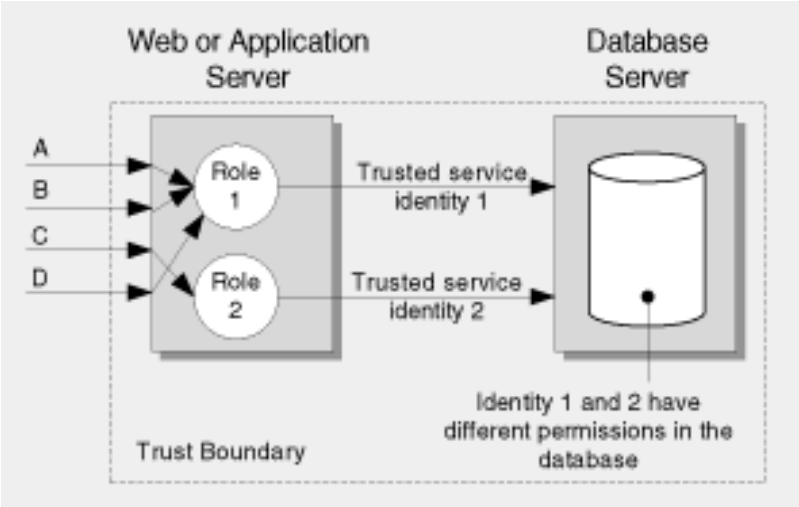

Figure 8. The multiple trusted service identities model

Network Infrastructure Security Considerations

Make sure 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, and therefore you should centralized 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

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 Sub-System

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** – Authenticate against a broker, which provides a token to use for authentication when accessing services or systems.
- **Direct Authentication** – Authenticate 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** – Implement a component that exposes an API that is different from the client API 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 Sub-System** – 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.

patterns & practices Solution Assets

- **Enterprise Library** provides a series of application blocks that simplify common tasks such as caching, exception handling, validation, logging, cryptography, credential management, and facilities for implementing design patterns such as Inversion of Control and Dependency Injection. For more information, see http://msdn2.microsoft.com/en-us/library/cc467894.aspx.
- **Unity Application Block** is a lightweight, extensible dependency injection container that helps you to build loosely coupled applications. For more information, see http://msdn.microsoft.com/en-us/library/cc468366.aspx.

Additional Resources

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

Presentation Technology Matrix

Objectives

- Understand the tradeoffs for each presentation technology choice.
- Understand the design impact of choosing a presentation technology.
- 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

Rich Client Applications

The following presentation technologies are suitable for use in Rich Client applications:

- **Windows Forms** – The standard UI design technology for the .NET Framework. Even with availability of WPF, it is still a good choice for UI design if your team is already has technical expertise with Windows Forms, and the application does not have any specific Rich UI requirements.
- **Windows Forms with WPF User Controls** – Allows you to take advantage of the more powerful UI capabilities provided by WPF controls. You can add WPF to your existing Windows Form application. Keep in mind that WPF controls tend to work best on higher-powered client machines.
- **WPF application** – Supports more advanced graphics capabilities such as 2D and 3D graphics, display resolution independence, advanced document and typography support, animation with timelines, streaming audio and video, and vector-based graphics. WPF uses XAML to improve the UI, data binding, and event definitions. WPF also includes advanced data binding and template capabilities. WPF applications can be deployed to the desktop or within a browser using XBAP. WPF applications support developer-designer interaction - developers can focus on the business logic, while designers can control the look and feel.
- **XAML Browser Application (XBAP) using WPF** – Hosts a sandboxed WPF application in Internet Explorer or Firefox on Windows. Unlike Silverlight, you can use most of the WPF framework but there are some limitations related to accessing system resources from the partial-trust sandbox. XBAP requires Windows Vista or both .NET 3.5 and the XBAP browser plug-in on the client desktop. XBAP is a good choice when the required features are not available in Silverlight, and you can specify the client platform and trust requirements.

- **WPF with Windows Forms User Controls** – Allows you to supplement WPF with controls that are not provided with WPF. You can use the WindowsFormsHost control provided in the WindowsFormsIntegration assembly to add Windows Forms controls. However, there are some restrictions or inconsistencies related to overlapping controls, interface focus, and rendering techniques used by the different technologies.

Benefits and Considerations Matrix

Rich Client Applications

Technology	Benefits	Considerations
Windows Forms	• Familiar programming model • Visual Studio designer support • Good performance on a wide range of client hardware	• Does not support 3D graphics, streaming media, flowable text, and other advanced UI features in WPF such as UI Styling and templates. • Does not support UI styling and templating. • Must be installed on the client
Windows Forms with WPF User Controls	• Allows you to add rich UI to existing Windows Forms applications • Provides a transition strategy to full WPF applications	• Depending on the complexity of your UI, may require higher-powered graphics hardware • You cannot overlay Windows Forms and WPF controls, reducing visual design flexibility
WPF application	• Rich UI and visualization including 2D graphics, 3D graphics, display resolution independence, vector graphics, flowable text, and animation • Supports variable bandwidth streaming media (Adaptive Media Streaming) • XAML makes it easier to define UI, data-binding and events • Supports separate developer/graphic designer integration	• Depending on the complexity of your UI, may require higher-powered graphics hardware • Your team may be less familiar with Expression Blend compared to Visual Studio • WPF ships with fewer built-in controls than Windows Forms
WPF with Windows Forms Controls	• Allows you to supplement WPF with controls that are not provided by WFP; for example, WPF does not provide a grid control	• Requires a WindowsFormsHost • May be difficult to get focus and input to transition across boundaries • You cannot overlap WPF and Windows Forms controls

		• WPF and Windows Forms controls use different rendering techniques, which can cause inconsistencies in how they look on different platforms
XAML Browser Application (XBAP) using WPF	• Allows you to deploy a WPF application to the Web • Provides all the rich visualization and UI benefits of WPF • Easier deployment and update than a WPF or Windows Forms application	• Only works on Vista or on a client with .NET 3.5 and the XBAP browser plug-in installed • Only works in Internet Explorer and Firefox browsers

Common Scenarios and Solutions

Rich Client Applications

Windows Forms

Consider using Windows Forms if:

- Your team already has experience building Windows Forms applications and you cannot afford to change to another technology.
- You are extending or modifying an existing Windows Forms application.
- You do not require rich media or animation support.

Windows Forms with WPF User Controls

Consider using Windows Forms with WPF user controls if:

- You already have a Windows Forms application and want to take advantage of WPF capabilities such as advanced graphics, flowable text, streaming media, and animations.

WPF

Consider using WPF if:

- You are building a rich client application and want to leverage the rich visualization and UI capabilities of WPF.
- You are building a rich client application that you may want to deploy to the web using XBAP.
- You want to use XAML to define your UI design, data binding, and events.
- You want to integrate the development process with graphic designers to who will specify the look and feel.

WPF with Windows Forms Controls

Consider using WPF with Windows Forms controls if:

- You are building a rich client application using WPF and want to use a control not provided by WPF.
- You are building a WPF application to leverage the rich visualization and UI capabilities.
- You want to use XAML to define your UI design, data binding, and events.

XAML Browser Application (XBAP) Using WPF

Consider using an XBAP that uses WPF if:

- You already have a WPF application that you want to deploy to the Web.
- You want to leverage rich visualization and UI capabilities of WPF that are not available in Silverlight.
- You are building a Web application for clients running Internet Explorer or Firefox that are guaranteed to have Windows Vista or .NET 3.5 with the XBAP browser plug-in installed.

Data Access Technology Matrix

Objectives

- Understand the tradeoffs for each data access technology choice.
- Understand the design impact of choosing a data access technology.
- 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 as well as 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 advantages and considerations of each one. Use the Common Scenarios and Solutions section to map your application scenarios to common data access technology solutions.

Data Access Technologies Summary

Following data access technologies are available with .NET platform

- **ADO.NET Core** – provides general retrieval, update, and management of data. ADO.NET includes providers for SQL Server, OLE-DB, ODBC, SQL Server Mobile, and Oracle databases.
- **ADO.NET Data Services Framework** – exposes data using the Entity Data Model, through RESTful Web services accessed over HTTP. The data can be addressed directly via a URI. The Web service can be configured to return the data as plain Atom and JSON formats.
- **ADO.NET Entity Framework** – gives you a strongly typed data access experience over relational databases. It moves the data model from the physical structure of relational tables to a conceptual model that accurately reflects common business objects. The Entity Framework introduces a common Entity Data Model (EDM) within the ADO.NET environment, allowing developers to define a flexible mapping to relational data. This mapping helps to isolate applications from changes in the underlying storage schema. The Entity Framework also contains support for LINQ to Entities, which provides LINQ support for business objects exposed through the Entity Framework. Current plans for the Entity Framework will build in functionality so it can be used to provide a common data model across high-level functions such as data query and retrieval services, reporting, synchronization, caching, replication, visualization, and BI. When used as an Object/Relational Mapping (O/RM) product developers use LINQ to Entities against business objects, which Entity Framework will convert to Entity SQL that is mapped against an Entity Data Model managed by Entity Framework. Developers also have the option of working directly with the Entity Data Model and using Entity SQL in their applications.
- **ADO.NET Sync Services –** is **a provider included i**n 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)** – provides class libraries that extend C# and Visual Basic with a native language syntax for queries. Queries can be performed against a variety of data formats, which include: DataSet (LINQ to DataSet), XML (LINQ to XML), In Memory Objects (LINQ to Objects), ADO.NET Data Services (LINQ to Data Services), and Relational data (LINQ to Entities). The main thing to understand is that LINQ is 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.
- **LINQ to SQL –** provides a lightweight strongly typed query solution against SQL Server. LINQ to SQL is designed for easy, fast object persistence scenarios where the classes in the mid-tier map very closely to database table structures. Starting with .NET 4.0, LINQ to SQL scenarios will be integrated and supported by ADO.NET Entity Framework, however LINQ to SQL will continue to be a supported technology. For more information see this post on ADO.NET team blog. http://blogs.msdn.com/adonet/archive/2008/10/31/clarifying-the-message-on-l2s-futures.aspx

Benefits and Considerations Matrix

Object-Relational Access

Technology	Benefits	Considerations
ADO.NET Entity Framework (EF)	• Decouples the underlying database structure from the logical data model. • Entity SQL (ESQL) provides a consistent query language across all data sources and database types. • Separates metadata into well-defined architectural layers. • Allows business logic developers to access the data without knowing database specifics. • Rich designer support in Visual Studio to visualize your data entity structure. • Provider model allows it to be mapped to many databases	• Requires you to change the design of your entities and queries if you are coming from a more traditional data access method. • You have separate object model. • More layers of abstraction than LINQ to DataSet. • Can be used with or without LINQ to Entities • If your database structure changes, you need to regenerate the Entity Data Model, and the EF libraries need to be redeployed.
LINQ to Entities	• LINQ based solution for relational data in the ADO.NET Entity Framework. • Provides strongly typed LINQ access to relational data	• Requires ADO.NET Entity Framework

	• Supports LINQ based queries against objects built on top of ADO.NET EF Entity Data Model. • Processing is on the server	
LINQ to SQL	• Simple way to read/write objects when object model matches database model. • Provides strongly typed LINQ access to SQL data. • Processing is on the server	• Functionality to be integrated into Entity Framework as of .NET 4.0 • Maps LINQ queries directly to the database instead of through a provider, and therefore works only with Microsoft SQL Server.

Disconnected and Offline

Technology	Benefits	Considerations
LINQ to DataSet	• Allows full-featured queries against a data-set	• Processing is all 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.

SOA / Service Scenarios

Technology	Benefits	Considerations
ADO.NET Data Services Framework	• Data can be addressed directly via URI using a REST-like scheme. • Data can be returned in either Atom or JSON formats. • Includes a lightweight versioning scheme to simplify the release of new service interfaces. • .NET, Silverlight and AJAX client libraries allow developers to work directly with objects and provide strongly typed LINQ access to Data Services • .NET, Silverlight and AJAX client libraries provide a familiar API surface to Windows Azure Tables, SQL Data Services and other Microsoft Services.	• Only applicable to service oriented scenarios.

LINQ to Data Services	• Allows you to create LINQ based queries against client-side data returned from ADO.NET Data Services. • Supports LINQ based queries against REST data.	• Can only be used with the ADO.NET Data Services client-side framework.

N-Tier

Technology	Benefits	Considerations
ADO.NET Core	• Includes .NET managed code providers for connected access to a wide range of data stores. • Provides facilities for disconnected data storage and manipulation.	• Code is written directly against specific providers, reducing reusability. • Relational database structure may not match the object model, requiring you to write a data mapping layer by hand.
ADO.NET Data Services Framework	• Simple out-of-the-box solution with ADO.NET Entity Framework • Data can be addressed directly via URI using a REST-like scheme. • Data can be returned in either Atom or JSON formats. • Includes a lightweight versioning scheme to simplify the release of new service interfaces. • Provider model allows any IQueryable data source to be used. • Data can be addressed directly via URI using a REST-like schema. • Data can be returned in either Atom or JSON formats. • Includes a lightweight versioning scheme to simplify the release of new service interfaces. • .NET, Silverlight and AJAX client libraries provide a familiar API surface to Windows Azure Tables, SQL Data Services and other Microsoft Services.	• Only applicable to service oriented scenarios. • Provides a resource-centric service that maps well to data heavy services, but may require more work if a majority of the services are operation centric.
ADO.NET Entity Framework	• Separates metadata into well-defined architectural layers. • Supports LINQ to Entities, for querying complex object models. • Provider model allows it to be mapped to many database types • Allows you to build services that have	• Requires you to change the design of your entities and queries if you are coming from a more traditional data access method. • Entity objects can be shipped across the wire, or you can use the Data Mapper pattern to transform entities into objects that are more generalized DataContract

	well defined boundaries, and data/service contracts for sending and receiving well defined entities across the service boundary • Instances of entities from your Entity Data Model are directly serializable and consumable by the web services • Full flexibility in structuring the payload – send individual entities, collections of entities or an entity graph to the server • Eventually will allow for true persistence ignorant (POCO) objects to be shipped across service boundaries	types. Planned addition of POCO support will eliminate the need to transform objects when shipping them across the wire. • Building service endpoints that receive generalized graph of entities is less "service oriented" than endpoints that enforce stricter contracts on the types of payload that might be accepted
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.
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 much easier to use. • Query results can be used as parameters to XElement and XAttribute object constructors.	• Relies heavily on generic classes. • Not optimized to work with untrusted XML documents, which require different mitigation techniques for security.
LINQ to SQL	• LINQ to SQL is a simple way to get objects in and out of the database when the object model and the database model are the same.	• As of .NET 4.0 the Entity Framework will be the recommended data access solution for LINQ to relational scenarios. • LINQ to SQL will continue to be supported and evolve based on feedback received from the community.

General Recommendations

- **Flexibility and Performance** – If you need maximum performance and flexibility, consider using ADO.NET Core. ADO.NET Core provides the most capabilities and is the most server-

specific solution. When using ADO.NET Core consider the tradeoff of additional flexibility vs. the need to write custom code. Keep in mind that mapping to custom objects will reduce performance. If you require a thin framework that uses the ADO.NET providers and supports database changes through configuration, consider the Data Access Application Block.

- **Object Relational Mapping(ORM)** – If you are looking for an ORM based solution and/or must support multiple databases, consider Entity Framework. This is ideal for implementing Domain Model scenarios.
- **Offline Scenario** – If you must support a disconnected scenario, consider using Datasets or Sync Framework.
- **N-Tier Scenario** – If you are passing data across layers or tiers, options available to you include passing entity objects, Data Transfer Objects (DTO) that are mapped to entities, DataSet and custom objects. If you are building resource-centric services (REST) consider ADO.NET data services. If you are building operation-centric services (SOAP) consider WCF services with explicitly defined service and data contracts.
- **SOA / Services Scenarios** – If you expose your database as a service, consider ADO.NET Data Services. If you would like to store your data in the cloud consider SQL Data Services.

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

Common Scenarios and Solutions

ADO.NET Core

Consider using ADO.NET Core if you:

- Need to use low level API for full control over data access your application.
- Want to leverage the existing investment made into ADO.NET providers.
- Are using traditional data access logic against the database.
- Do not need the additional functionality offered by the other data access technologies.
- Are building an application that needs to support disconnected data access experience.

ADO.NET Data Services Framework

Consider using ADO.NET Data Services Framework if you:

- Are developing a Silverlight application and want to access data through a data centric service interface.
- Are developing a rich client application and want to access data through a data centric service interface.
- Are developing N-tier application and want to access data through data centric service interface.

ADO.NET Entity Framework

Consider using ADO.NET Entity Framework (EF) if you:

- Need to share a conceptual model across applications and services.

- Need to map a single class to multiple tables via Inheritance.
- Need to query relational stores other than the Microsoft SQL Server family of products.
- Have an object model that you must map to a relational model using a flexible schema.
- Need the flexibility of separating the mapping schema from the object model.

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.

LINQ to Data Services

Consider using LINQ to Data Services if you:

- Are using data returned from ADO.NET Data Services in a client.
- Want to execute queries against client-side data using LINQ syntax.
- Want to execute queries against REST data using LINQ syntax.

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 Entities

Consider using LINQ to Entities if you:

- Are using the ADO.NET Entity Framework
- Need to execute queries over strongly-typed entities.
- Want to execute queries against relational data using LINQ syntax.

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 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 LINQ syntax.

LINQ to SQL Considerations

LINQ to Entities is the recommended solution for LINQ to relational database scenarios. LINQ to SQL will continue to be supported but will not be a primary focus for innovation or improvement. If you are already relying upon LINQ to SQL you can continue using it. For new

solutions, consider using LINQ to Entities instead. At the time of this writing, this is the product group position:

"We will continue make some investments in LINQ to SQL based on customer feedback. This post was about making our intentions for future innovation clear and to call out the fact that as of .NET 4.0, LINQ to Entities will be the recommended data access solution for LINQ to relational scenarios."

For more information see the ADO.NET team blog.
http://blogs.msdn.com/adonet/archive/2008/10/31/clarifying-the-message-on-l2s-futures.aspx

Additional Resources

For more information, see the following resources:

- *ADO.NET* at http://msdn.microsoft.com/en-us/library/e80y5yhx(vs.80).aspx.
- *ADO.NET Data Services* at http://msdn.microsoft.com/en-us/data/bb931106.aspx.
- *ADO.NET Entity Framework* at http://msdn.microsoft.com/en-us/data/aa937723.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 - Rich Client Application

Design Considerations

- Design patterns, such as MVC and Supervising Controller, are used to separate presentation logic from user interface implementation.
- User interface features such as layout, navigation, choice of controls, and localization are designed to maximize accessibility and usability.
- Business rules and other tasks not related to the interface are extracted to a separate business layer.
- Client is loosely coupled from remote services it uses.
- Unnecessary round trips are reduced when accessing remote layers.

Communication

- Communication protocols and ports for the application are supported and allowed in your network infrastructure.
- Message-based protocols are used to communicate with business layers, services, and components on a remote physical tier when possible.
- Sensitive information is protected during network transfer through use of security conscious protocols and transfer methods such as IPSec, SSL, and digital signatures
- Application enables offline processing by detecting the connection state, caching information locally, and then re-synchronizing when communication is re-enabled.
- Coarse-grained interfaces are used when communicating with business layers, services, and components on a remote physical tier to minimize network traffic and maximize performance.

Composition

- Appropriate types of interface components have been identified based on functional specifications and requirements.
- You have considered user interface composition patterns like View Injection and View Discovery
- A consistent appearance is maintained across all interface components so as not to confuse the user.
- A Versioning strategy has been designed to version and manage updates of UI components
- Dynamic loading is utilized to allow easy replacement of components and reduce the impact of component changes and updates.

Configuration Management

- Configurable data that may change in the life of the application is identified.
- The storage location has been identified, whether it should be stored locally or retrieved from elsewhere within the application layers..

- Global application settings are designed to be stored in a central location.
- Sensitive configuration information is protected when in memory, stored locally and in transit over the network.
- The design takes into account any global security policies that may affect or override local configurations.

Data Services

- Translation is used to handle data formats that are not supported by the client application.
- UI responsiveness maximized by loading data asynchronously when possible.
- A service dispatcher mechanism is used to monitor connectivity and send batched updates when a suitable connection is available.
- You have researched existing frameworks and tools that can reduce development time and effort for general data access scenarios. For example, Microsoft Data Services, Microsoft ADO.NET Entity Framework, ADO.NET LINQ, and the Enterprise Library Data Access Application Block.
- Large data sets are chunked and loaded asynchronously into a local cache to improve performance.

Exception Management

- Exceptions are not used to control logic flow.
- Exceptions are caught only if they can be handled.
- Global error handler is used to catch unhandled exceptions.
- User friendly messages are displayed when an exception or error occurs in the system.
- Exception details do not reveal sensitive information.

State Management

- The caching mechanism is designed considering the state information that the application must cache including estimates of the size, the frequency of changes, and the processing or overhead cost of recreating or re-fetching the data.
- The application uses local disk-based caching mechanism for storing large volumes of data.
- User's preferences and configuration settings for the application is cached in memory to improve performance.
- Data which needs to be available at application start-up is stored using a persistent caching mechanism such as Isolated Storage or a disk file.
- Large amount of data which is retrieved from another physical tier and not required when the application start-up is loaded into cache asynchronously.
- Sensitive cached data is protected using encryption and digital signatures.

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..

Presentation Considerations

- The UI is implemented for all the user and business application requirements.
- Design patterns, such as Command, Publish/Subscribe, and Observer are used to decouple commands and navigation from the components in the application and to improve testability.
- Workflows or Viewflows are used for implementing multi-step Wizard-like interface elements.
- Data-binding capabilities are used to display data, whenever possible. Two-way binding is used where the user is able to update the data.
- The application is designed to support globalized and localization.
- The data used by the UI is separated from the UI itself to improve testability.

Business/Service Layer Considerations

- The business layer and service interfaces are identified and the business layer service functions are designed to be accessed using the interfaces.
- The business logic which does not contain sensitive information is located on the client to improve performance of the UI and the client application.
- The business layer which contains sensitive information is located on separate tier.
- The client is designed to obtain information required to operate business rules and other client-side processing, and to update business rules automatically as requirements change.

Maintainability Considerations

- A suitable mechanism for manual and/or automatic updates to the application and its components is implemented taking into account versioning issues to ensure that the application has consistent and interoperable versions of all the components it uses.
- The appropriate deployment option is chosen depending on the presentation technology used, these may include ClickOnce, Windows Installer, and XCOPY.
- Components are designed to be interchangeable where possible, allowing change to individual components depending on requirements, runtime scenarios, and individual user requirements or preferences.
- Appropriate logging and auditing mechanism is implemented for the application to assist administrators and developers when debugging the application and solving runtime problems.
- Components and layers have minimal dependencies so that the application can be used in different scenarios where appropriate, without changes to other layers affecting the client application.

Security Considerations

- Appropriate authentication strategy is designed for disconnected or offline authentication where this is relevant.
- Appropriate authorization strategy is designed and user identity flown to backend services is protected.
- Single-Sign-On (SSO) or federated authentication solution is designed to access multiple applications with the same credentials or identity.
- Appropriate validation strategy is designed to validate inputs, both from the user and from sources such as services and other application interfaces.
- Sensitive data is encrypted where it may be exposed, especially over a network or communication channel, and you have considered using a digital signature to prevent tampering. In maximum security applications, you have considered encrypting volatile information stored in memory.

Offline / Occasionally Connected Considerations

- Asynchronous communication is used whenever possible for interacting with data and services over a network.
- Complex interactions with network-located data and services are minimized or eliminated.
- Data caching is designed to ensure that all of the data necessary for the user to continue working is available on the client when it goes offline.
- A mechanism to manage connections is designed that takes into account the environment in which your application operates, both in terms of the available connectivity and the desired behavior of your application as this connectivity changes.
- A store-and-forward mechanism is designed for communication so that messages are created, stored while the application is disconnected, and forwarded to their respective destinations when a connection becomes available.
- Data refresh mechanism is designed to deal with stale cached data, and preventing the Rich Client from using stale data.

Deployment Considerations

- The target physical deployment environment is identified, early in the planning stage of the design lifecycle.
- Environmental constraints that may impact your software design and architecture decisions are identified.
- Aspects of the software design that require certain infrastructure attributes are identified.

Checklist - Presentation Layer

Design Considerations

- UI technology choice is based on application requirements and constraints; for example, broad reach across platforms and browsers.
- Relevant patterns 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.
- 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

Composition

- Dynamically-loaded, reusable views are used to simplify the design, improve performance, and increase maintainability.
- The Dependency Injection pattern is used to support dynamic loading and replacement of modules.
- The Composite View pattern is used if you need to compose views from modular, atomic components.
- The Template View pattern, through the use of Master Pages, is used to create consistent, reusable, dynamic web pages.
- The Publish/Subscribe pattern is used for communication between dynamically loaded modules.
- The Command pattern is used to support menu and command-driven interaction.

Exception Management

- Sensitive data or internal application details are not revealed to users in error messages or in exceptions that cross trust boundaries.
- 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.
- The layout has been optimized for search engines.
- Cascading Style Sheets (CSS) are used wherever possible for layout.

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.

Presentation Entities

- Presentation entities are used only if you need to manage unique data or data formats in the presentation layer.
- Presentation entities do not contain business logic.
- Custom classes are used to map data directly to business entities.
- Platform-provided classes, such as DataSets or Arrays, are used for data-bound controls.
- Presentation entities contain input validation logic for the presentation layer.

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.
- AJAX is used if user responsiveness is important.
- 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.
- State is stored in platform provided state management features, such as ViewState, for standard ASP.NET 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 or MVP pattern.
- UI processing components do not include business rules.

Validation

- The application constrains, rejects and sanitizes all input that comes from the client.
- Server-side validation is used to validate input for security purposes.
- Client-side validation is used to validate input for user experience purposes; for example, to provide error messages when receiving invalid input.
- 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.
- Design reduces round trips when accessing a remote business layer.
- Business layer is not tightly coupled to other layers.

Authentication

- Users are authenticated in the business layer, unless they come from another layer on the same tier to which you are willing to extend full trust.
- Single-sign-on is used, if your business layer will be used by multiple applications in a trusted environment.
- The original caller is not flowed to the business layer, unless it is necessary to authenticate based on the original caller's ID.
- A trusted sub-system is used for access to back-end services to maximize the use of pooled database connections.
- IP filtering is used when using Web services, to only allow calls from the presentation layer

Authorization

- Users are authorized based on their identity, account groups, claims or roles.
- Role-based authorization is used for business decisions.
- Resource-based authorization is used for system auditing.
- Claims-based authorization is used to support federated authorization.
- Impersonation and delegation are not used unless absolutely necessary and the performance trade-offs are well understood.

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 Domain Model pattern is used for designing business entities, if the application needs to support complex business model.

- 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.
- Web farm deployment scenario considered, while designing business layer caching solution.

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.

Service Interface

- The service interface is abstracted from potential internal changes.
- An interface exists for each client access scenario.
- The service interface does not implement business rules.
- Standard data types are used as interface parameters to enable maximum compatibility with different clients.
- Service interfaces are designed, for maximum interoperability with other platforms and services.

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.

Deployment Considerations

- Business logic is deployed on a physically separate machine, only if it is actually required.
- Message-based interface is used for the business layer if the business layer is to be deployed on remote tier.
- TCP protocol is used if the business layer must be on a separate physical tier.
- IPSec is used to protect data passed between physical tiers.
- SSL is used to protect data passed to remote Web services.

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 disk.
- 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
- You do not store BLOB in the database when using Buffered transmission.

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

- Connections are opened as late as possible and closed as early as possible.
- Trusted sub-system authentication was used to maximize the advantages of connection pooling.
- 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.

Stored Procedures

- Output parameters are used to return single values.
- Individual parameters are used for single data inputs.
- XML parameters have been considered for passing lists or tabular data.
- Memory-based temporary tables are used when required.
- Error handling has been implemented to return errors that can be handled by the application code.

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.
- Automatic or implicit transactions are used when a transaction spans multiple databases.
- You have considered the use of Multiple Active Result Sets (MARS) in transaction heavy concurrent applications to avoid potential deadlock issues.

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.
- All trust boundaries are identified, and data that crosses these boundaries is validated.

- 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.
- An XML schema is used to define formats for data stored and transmitted as XML.
- XML data is validated against the appropriate schemas.
- 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.
- You have considered the use of stored procedures to abstract data access from the underlying data schema.

Performance Considerations

- Connection pooling has been optimized based on performance testing.
- 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 instead of SQL authentication when using Microsoft SQL Server.
- Encrypted connection strings in configuration files are used instead of a System or User DSN.
- 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.

Deployment Considerations

- The data access layer is located on the same tier as the business layer to improve application performance.
- The TCP protocol is used to improve performance when you need to support a remote data access layer.
- The data access layer is not located on the same server as the database.

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.