Microsoft

Application Architecture Guide 2.0

Designing Applications on the .NET Platform

patterns & practices

Application Architecture Guide 2.0

patterns & practices

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

Foreword by S. Somasegar

In using our own technologies to build Microsoft products, and working with customers and partners every day, we have developed practical guidance on how to use our technologies and application architecture that is valuable to the Developer and IT Professional community. We have built the *Application Architecture Guide 2.0* to consolidate guidance that we have gathered from our internal practices, external experts, customers, and others in the community to share with you.

This guide is a consolidated set of principles, patterns, and practices for designing application architecture. The purpose of the guide is to help solution architects and developers design and build more effective applications on the .NET platform and support top-level decision-making at the early stages of a new project, as well as provide topic-specific content to help architects and developers improve their existing solutions. This guidance incorporates the contributions and reviews of over fifteen external experts and customers.

The Architecture Meta Frame is a conceptual framework used throughout the guide. By thinking about applications in terms of scenarios, application types, architecture styles, and requirements, you can very quickly determine relevant technologies, patterns, and solution assets. You can then use the Architecture Frame as an additional lens to identify potential hotspots in your application architecture. For example, by focusing on data access or validation, you can prototype and test potential solutions to figure out the best approach for your scenario.

The guide includes a reference application architecture; common canonical application archetypes for Web, rich client, RIA, mobile, and services applications; architecture frames; quality attributes; and a design approach to help you design your architecture.

To summarize, we are confident that *Application Architecture Guide 2.0* will help you choose the right architecture, choose the right technologies, make more effective design decisions, and choose relevant patterns.

Sincerely,

S. Somasegar
Senior Vice President of Developer Division
Microsoft

Foreword by Scott Guthrie

Application architecture is a challenging topic, as evidenced by the wide variety of books, articles, and white papers on the subject. It is still too hard for developers and architects to understand architecture and design best practices for the .NET platform. The original *Application Architecture for .NET: Designing Applications and Services* guide did a great job of covering this topic, but it was written in 2002.

To deal with the many technology additions since then, J. D. Meier and his team from Microsoft patterns & practices have created a new application architecture guide to provide insightful guidance for designing .NET applications based on the latest practices and technologies. The outcome is *Application Architecture Guide 2.0*, a guide targeted to help solution architects and developers design effective applications on the .NET platform. The guide gives an overview of the Microsoft .NET platform and the main technologies and capabilities within it. It provides not only principles-based guidance, but also platform-independent, pattern-oriented guidance for designing your application. Even though the guide itself is comprehensive on its own, the team has also provided a Knowledge Base on the Web, which provides informative expansion on the topics and concepts outlined in the guide.

The guide is based on a number of frames that provide structure as it dissects the .NET technology space. The Architecture Frame provides a set of common categories to cover hotspots for key engineering decisions. The Quality Attributes Frame provides cross-cutting qualities and abilities that shape your application architecture, such as performance, security, scalability, manageability, deployment, communication, and more. The Canonical Application Frame describes, at a meta-level, the tiers and layers that an architect should consider. Each tier/layer is described in terms of its focus, function, capabilities, common design patterns, and technologies. Using each of these frames as a backdrop, the guide then overlays relevant principles, patterns, and practices. Finally, the guide provides canonical application archetypes to illustrate common application types. Each archetype is described in terms of the target scenarios, technologies, patterns, and infrastructure it contains.

The guidance as a whole is based on the combined experience and knowledge of Microsoft experts, Microsoft partners, customers, and others in the community. It will help you understand our platform, choose the right architecture, choose the right technologies, and build applications using proven practices and lessons learned.

Sincerely,
Scott Guthrie
Corporate Vice President of .NET Developer Platform
Microsoft

Introduction

Overview

The purpose of the Application Architecture Guide 2.0 is to improve your effectiveness when building applications on the Microsoft platform. The primary audience for this guide is solution architects and development leads. The guide provides design-level guidance for the architecture and design of applications built on the Microsoft .NET platform. It focuses on the most common types of applications and on partitioning application functionality into layers, components, and services, and also walks through their key design characteristics.

The guidance is task-based and presented in parts 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 is divided into the following four parts:

- **Part I, "Fundamentals,"** provides you with the fundamentals you will need in order to understand architecture design techniques and strategies.
- **Part II, "Design,"** provides overarching design guidelines and practices that can be applied to any application type or application layer, including guidelines on how to design a communications solution and plan for services.
- **Part III, "Layers,"** provides architecture and design approaches, as well as practices for each layer, including the presentation, business, service, and data access layers.
- **Part IV, "Archetypes,"** provides patterns and design frames for each application archetype; including service applications, Web applications, rich client applications, rich Internet applications (RIA), and mobile applications.

Why We Wrote This Guide

We wrote this guide to help you:

- Design more effective architectures on the .NET platform.
- Choose the right technologies for your particular scenario.
- Make more effective choices for key engineering decisions.
- Map appropriate strategies and patterns.
- Map relevant patterns & practices solution assets.

Scope

This guide provides principles, patterns, and practices for designing application architectures on the .NET platform. The guide presents a principle-based approach. The overall scope of the guide is shown in Figure 1.

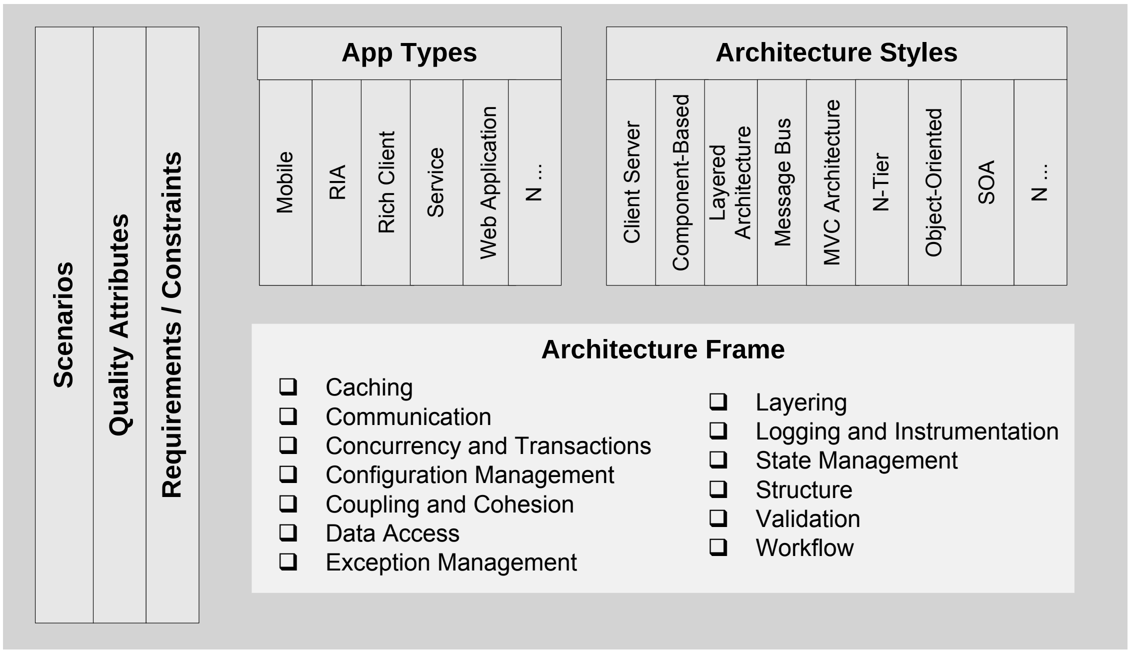

Figure 1 Organization of the Guide

The guidance is organized by application types, frames, layers, and quality attributes as follows:

- **Canonical application frame**. This frame describes, at a meta-level, the tiers and layers that an architect should consider. Each tier/layer is described in terms of its focus, function, capabilities, common design patterns, and technologies.
- **Application types**. Canonical application archetypes are used to illustrate common application types. Each archetype is described in terms of the target scenarios, technologies, patterns, and infrastructure it contains. Each archetype is mapped to the canonical application frame. The archetypes are illustrative of common application types, and are not comprehensive or definitive.
- **Arch frame**. This frame is a common set of categories of hotspots for key engineering decisions.
- **Quality attributes**. This is a set of qualities/abilities that shape your application architecture, such as performance, security, scalability, manageability, deployment, and communication.
- **Principles, patterns, and practices**. Using the frames as backdrops, the guide overlays relevant principles, patterns, and practices.
- **Technologies and capabilities**. This is a description/overview of the Microsoft custom application development platform and its main technologies and capabilities.

Features of This Guide

- **Framework for application architecture**. The guide provides a framework that helps you to focus your thinking 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 only the parts you need.
- **Holistic**. If you do read the guide from beginning to end, it is organized to fit together in a logical sequence. 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, and 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. Each recommendation is then expanded to help you understand what to do, why, and how.
- **Fast Track**. This section takes a fast path through the essentials of the framework used by the guide to help you quickly implement the guidance in your organization.

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.

What to Expect from This Guide

This guide is not a step-by-step tutorial for architecture and design, but rather is a reference. It provides a frame for hotspots within the application architecture space. The frame serves as a durable and evolvable backdrop for key principles, patterns, and practices. While the guide is comprehensive, it is not complete. Where possible, the guide serves as a map of the space rather than an elaboration on a particular point. The guide does not aim at providing a single, comprehensive end-to-end solution to a problem but instead provides concise recommendations to some of the most important problems you might encounter. The bulk of the guide is technology-agnostic and principled-based, but we have added technology considerations where we think it helps you to either choose among available technologies or make the most of them.

How to Use This Guide

You can read this guide from beginning to end, or you can read only the relevant parts or chapters. You can adopt the guide in its entirety for your organization, or you can use critical components to address your highest-priority needs. If you need to move quickly, use the Fast Track section. If you have more time and want to fully understand the architecture and design approach, you can work the guidance into your application development life cycle and processes, and use it as a training tool.

Ways to Use the Guide

You can use this comprehensive guidance in several ways, both to learn more about the architectural process and as a way to instill knowledge in the members of your team. Consider using the guide in the following ways:

- **Use it as a reference**. Use the guide as a reference for learning the architecture and design practices for 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 by many experienced 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.
- **Perform architecture and design reviews**. Use the question-driven approach to evaluate architecture and design choices. Use the guidance as a starting point, modify the list of questions to suit your requirements, and expand the list as you learn more.
- **Create training**. Create training based on the concepts and techniques used throughout the guide, as well as from its technical insight into the .NET Framework technologies.

Organization of This Guide

This guide is arranged into parts and chapters. Each part maps to a key architecture and design focus area. Each chapter provides actionable guidance on how to architect and design your application. Guidelines and lessons learned are summarized using bulleted lists, and are presented as practices for each area. Special chapters such as “Fast Track,” “Fundamentals of Application Architecture,” and “Architecture Styles” help you to apply the guidance more quickly and more easily.

Architecture and Design Solutions at a Glance

The “Architecture and Design Solutions at a Glance” section provides a problem index for the guide, highlighting key areas of concern and indicating where to look for more details.

Fast Track

The “Fast Track” section at the front of the guide helps you to implement the recommendations and guidance quickly and easily.

Part I, “Fundamentals”

This part introduces the fundamental application architecture concepts and terminology. It also helps you to understand the architecture design techniques and strategies. Part I includes the following chapters:

- Chapter 1, “Fundamentals of Application Architecture”
- Chapter 2, “.NET Platform Overview”
- Chapter 3, “Architecture and Design Guidelines”

Part II, “Design”
This part provides an approach to architecture design and discusses key architecture decisions such as deployment, architecture style, quality attributes, and communication options. Part II includes the following chapters:
- Chapter 4, “Designing Your Architecture”
- Chapter 5, “Deployment Patterns”
- Chapter 6, “Architecture Styles”
- Chapter 7, “Quality Attributes”
- Chapter 8, “Communication Guidelines”

Part III, “Layers”
This part provides architectural and design approaches, as well as practices, for each layer, including the presentation, business, service, and data access layers. Part III includes the following chapters:
- Chapter 9, “Layers and Tiers”
- Chapter 10, “Presentation Layer”
- Chapter 11, “Business Layer Guidelines”
- Chapter 12, “Data Access Layer Guidelines”
- Chapter 13, “Service Layer Guidelines”

Part IV, “Archetypes”
This part provides patterns and design frames for each application archetype; including service applications, Web applications, rich client applications, rich Internet applications, and mobile applications. Part IV includes the following chapters:
- Chapter 14, “Application Archetypes”
- Chapter 15, “Web Applications”
- Chapter 16, “Rich Internet Applications (RIA)
- Chapter 17, “Rich Client Applications”
- Chapter 18, “Services”
- Chapter 19, “Mobile Applications”
- Chapter 20, “Office Business Applications (OBA)”
- Chapter 21, “SharePoint Line-Of-Business (LOB) Applications”

Approach Used in This Guide
How do you design successful applications on the .NET platform? This guide describes an approach that starts with an understanding of the entire architectural process, and then focuses in on the specific topics, techniques, practices, and application types to help you

achieve the desired results for your design. The suggested approach for using this guide is as follows:

- If you are using the guide to learn about the fundamentals of application architecture, start with Part I, "Fundamentals," and then work through the remaining parts to learn about design, layers, quality attributes, and archetypes.
- If you have some prior experience with application architecture, start with Part II, "Design," to understand modern design approaches for the major features of an application, services, and communication techniques. Then use the remaining sections to expand and verify your knowledge.
- If you want to learn about the layered approach to application architecture, start with Part III, "Layers," and then explore the following sections to see how to apply quality attributes and how to design specific types of applications.
- If you are experienced in application architecture but want to learn about the different types of application that you can build and the specific features of each type, look at Part IV, "Archetypes."
- Alternatively, for a brief overview of the architectural process, read the "Architecture and Design Solutions at a Glance" and "Fast Track" chapters, and then use the remaining sections of the guide to gain specific additional knowledge.

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 Knowledge Base at http://www.codeplex.com/AppArchGuide.

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 Microsoft MSDN® Newsgroups site at http://msdn.microsoft.com/en-us/subscriptions/aa974230.aspx.

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

Many thanks to the contributors and reviewers:

- **Test Team**. Rohit Sharma; Praveen Rangarajan
- **Edit Team**. Dennis Rea
- **External Contributors and Reviewers**. Adwait Ullal; Andy Eunson; Brian Sletten; Christian Weyer; David Guimbellot; David Ing; David Weller; Derek Greer; Eduardo Jezierski; Evan Hoff; Gajapathi Kannan; Jeremy D. Miller; John Kordyback; Keith Pleas; Kent Corley; Mark Baker; Paul Ballard; Peter Oehlert; Norman Headlam; Ryan Plant; Sam Gentile; Sidney G Pinney; Ted Neward; Udi Dahan
- **Microsoft Contributors and Reviewers**. - Ade Miller; Amit Chopra; Anna Liu; Anoop Gupta; Bob Brumfield; Brad Abrams; Brian Cawelti; Bhushan Nene; Burley Kawasaki; Carl Perry; Chris Keyser; Chris Tavares; Clint Edmonson; Dan Reagan; David Hill; Denny Dayton; Diego Dagum; Dmitri Martynov; Dmitri Ossipov; Don Smith; Dragos Manolescu; Elisa Flasko; Eric Fleck; Erwin van der Valk; Faisal Mohamood; Francis Cheung; Gary Lewis; Glenn Block; Gregory Leake; Ian Ellison-Taylor; Ilia Fortunov; J.R. Arredondo; John deVadoss; Joseph Hofstader; Koby Avital; Loke Uei Tan; Luke Nyswonger; Manish Prabhu; Meghan Perez; Mehran Nikoo; Michael Puleio; Mike Francis; Mike Walker; Mubarak Elamin; Nick Malik; Nobuyuki Akama; Ofer Ashkenazi; Pablo Castro; Pat Helland; Phil Haack; Reed Robison; Rob Tiffany; Ryno Rijnsburger; Scott Hanselman; Seema Ramchandani; Serena Yeoh; Simon Calvert; Srinath Vasireddy; Tom Hollander; Wojtek Kozaczynski

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.

Architecture and Design Solutions at a Glance

Objectives

- Learn about the typical general approach to designing application architectures.
- Learn about the different types of applications that you can build.
- Learn about the key factors you must consider when designing application architectures.
- Learn about the layered approach to application design.
- Learn about the quality attributes that affect the overall success and acceptance of an application.

Overview

This chapter presents a wide-ranging overview of the key factors involved in designing the architecture for your applications. It describes the general steps in the process, the types of applications you can build, the typical layered architecture used for applications, and the design approaches for data access, services, and quality attributes.

General

The following guidelines will help you to understand some of the fundamental techniques to use when designing your application. Use these guidelines as a starting point toward understanding how to design your application, how to structure your application, how to use layers, and how to review your design.

- **How to design your architecture**
  First, determine the objectives of your architectural design, which will help you to focus on solving the right problems. Then identify the key scenarios or use cases that are most critical from your application's perspective and that have maximum impact on the architecture. The use cases will help you to identify critical problem areas and will also help in evaluating and reviewing the architecture. To achieve the architectural objectives, identify the application type, deployment architecture, architecture style, and related technologies in view of the identified key scenarios. While doing so, ensure that you take into account any infrastructural and organizational constraints. Identify the quality attributes (performance, security, etc.) and cross-cutting concerns (exception handling, auditing and logging, and so on) that are important from an application perspective, to make sure that the architecture adheres to their requirements and objectives. Finally, define the architecture solution and review/validate it against the key scenarios you identified.

  For more information, see Chapter 4, "Designing your Architecture."

- **How to choose a deployment topology**
  When you design your application architecture, you must take into account corporate policies and procedures, together with the infrastructure on which you plan to deploy your application. If the target environment is rigid, your application design must reflect the restrictions that exist in that rigid environment. Your application design must also take into account quality attributes such as performance, security, and maintainability. Sometimes you must make design tradeoffs because of protocol restrictions and network topologies. Identify the requirements and constraints that exist between application architecture and infrastructure architecture early in the development process. Use a layered design that separates presentation, business, and data access logic, to improve scalability and maintainability. To avoid unnecessary remote calls and additional network latency, stay in the same process where possible. If you do need a distributed architecture, consider the implications of remote communication when you design your interfaces. For example, you might need a distributed architecture because security policy prevents you from running business logic on your Web server, or you might need a distributed architecture because you need to share business logic with other applications, Try to reduce round trips and the amount of traffic that you send over the network.

  For more information, see Chapter 5, "Deployment Patterns."

- **How to structure your application**
  Start at the highest level of abstraction and begin by grouping your application functionality into logical layers. Understand the available deployment strategies for the physical structure and tiers, and identify how the logical layers will map to the physical tiers. Identify communication strategy and the protocols that can be used for communication between the layers and tiers in your application. Define interfaces for the layers, the information that will be communicated between layers, and the form of that communication.

  For more information, see Chapter 3, "Architecture and Design Guidelines."

- **How to decide on your layering strategy**
  When grouping the components in a layer, make sure that the components depend only on components in the same layer or components in a lower layer. Use loose coupling between layers. Avoid any circular dependencies between layers. Use only the layers that are required and that are relevant to your scenario. For example, if your application does not have a user interface (UI), you will not require a presentation layer, but only a programmatic (service) interface. If your application has minimal or no business logic (for example, a reporting application), you might not require a business layer. If your business logic is not shared by other applications, or is not located on a remote tier, you might not require a service layer. If your application does not use an external data source, you might not require a data access layer.

  For more information, see Chapter 9, "Layers and Tiers."

- **How to perform architecture and design reviews**
  Identify the architecture objectives and key scenarios for your applications that have the greatest impact on the application architecture. Establish design and architecture standards for your application; use these standards to perform design and architecture inspections, using a question-driven approach with respect to architecture objectives, infrastructure and organizational constraints, performance, and security goals, to focus your efforts. Use the key scenarios identified for scenario-based evaluations, such as Software Architecture Analysis Method (SAAM), Architecture Tradeoff Analysis Method (ATAM), and so on.

  For more information, see Chapter 4, "Designing Your Architecture."

Application Types

The following guidelines will help you to understand the fundamental factors you must consider when choosing an application type. Use these guidelines as a starting point toward understanding how to choose an application type, and the key differences between each application type covered in this guide. For example, device resources and performance are key considerations when designing a mobile application; choice of UI design patterns is a key consideration when designing a rich client application; and changes to your UI design and deployment paradigms are key considerations when designing a rich Internet application. Key considerations when designing a service interface include choosing a pattern for effective decoupling. Key considerations when designing a Web application include layering, as well as the amount of client-side processing to use.

- **How to choose an application type**
  Your choice of application type will be driven primarily by the scenarios that you want to support. If you want to leverage client resources and support disconnected scenarios, consider a rich client application. If you want to provide the application UI over the Web, consider building a Web application. If you want to support advanced graphics and streaming media in a Web-deployed scenario, consider a rich Internet application (RIA). If you want to expose a loosely coupled interface to remove clients without a UI, consider building a service. If you want to support mobile devices over the Internet, consider designing a mobile Web application. If you want to support mobile devices in which you leverage device resources or need to support partially disconnected scenarios, consider a mobile rich-client application.

  For more information, see Chapter 14, "Application Archetypes."

- **How to design a Web application**
  Most of the code associated with a Web application resides on a Web server, with some code implemented in the client browser. The design you use on the Web server can vary, based on the type and size of the application you are building. For smaller applications with few or no business rules, a client/server design can be used where presentation code that generates HTML interacts directly with data-access components. For most business

applications, or larger applications that contain business rules, consider using a layered design that separates presentation, business, and data-access functionality into separate layers. In addition, the interaction between the presentation and business layers should be message-based, which is better suited to the stateless nature of Web application requests. On the client, you can improve the user experience by using dynamic HTML (DHTML), JavaScript, Asynchronous JavaScript and XML (AJAX), or a combination of all three technologies. For example, with AJAX you can implement lazy loading and partial updates to page content, which will improve performance and prevent the “reload flash” when a user interacts with the page.

For more information, see Chapter 15, “Web Applications.”

- **How to design a rich client application**
  For reusability and testability, separate the presentation logic from the UI implementation by using a design patterns such as Model-View-Controller (MVC). Design to provide a suitable and usable interface in terms of layout, navigation, choice of controls, and localization. Extract business rules and other tasks not related to the UI into a separate business layer. Use a message-based interface to communicate with services deployed on separate physical tiers. Avoid tight coupling to objects in other layers by using common interface definitions, abstract base classes, or message-based communication. For example, implementing the Dependency Injection and Inversion of Control patterns can provide a shared abstraction between layers.

  For more information, see Chapter 17, “Rich Client Applications.”

- **How to design a rich Internet application (RIA)**
  Plan to use a Web-based infrastructure, because RIA implementations require an infrastructure similar to that of Web applications. Design your application to run in the browser sandbox. When designing an RIA, consider using a single page that changes dynamically as the user works with the application. Multi-page designs are more complex in an RIA, and require additional considerations such as deep linking and UI screen navigation. Design for usability, such as the ability to pause and navigate to the appropriate point in a workflow without restarting the whole process. RIA implementations have a small footprint on the client, but require a browser plug-in. Design for scenarios in which the browser plug-in is not already installed, including non-interruptive plug-in installation and displaying informative error messages if an installation problem should occur.

  For more information, see Chapter 16, “Rich Internet Applications.”

- **How to design a service**
  When designing a service, there are general guidelines that should be followed; for example, design for extensibility, use coarse-grained interfaces, never assume how the client will use the service, and decouple the interface from the implementation. The best way to support these guidelines is to introduce a service layer in your design that sits

between consumers of the service and the business layer that supports the service. Within the service layer, you define interface contracts, classes to translate between the interface and the business layer, and concrete classes that implement the interface. In most cases, interaction with the business layer is accomplished by using a Façade pattern, which allows you to combine multiple business operations into a single application-scoped service operation.

For more information, see Chapter 18, "Services."

- **How to design a mobile application**
  Design specifically for the device instead of trying to reuse the architecture or UI from a desktop application or a Web application. When choosing which device types to support, consider screen size and format, CPU performance characteristics, memory and storage space, development tool and environment support, as well as user requirements and organizational constraints. Design your caching, state management, and data-access mechanisms with intermittent network connectivity in mind.

  For more information, see Chapter 19, "Mobile Applications."

Architecture Frame

The following guidelines will help you to understand the fundamental cross-cutting factors you must consider when designing your chosen application type. Use these guidelines as a starting point toward understanding how to think about key functionality that cuts across layers and tiers. For example, your exception-management strategy should be designed to be consistent across your entire application, and not with just a single layer in mind.

- **How to design your exception-management strategy**
  Use structured exception handling to build robust code. Use exceptions instead of error codes where possible. Do not reveal internal system or application details, such as stack traces, SQL statement fragments, and so on. Ensure that this type of information is not allowed to propagate to the end user, or beyond your current trust boundary. Fail securely in the event of an exception, and make sure that your application denies access and is not left in an insecure state. Use **finally** blocks to guarantee that resources are cleaned up when exceptions occur; for example, close your database connections in a **finally** block. Do not log sensitive or private data such as passwords, which could be compromised. When the exception contains user input in the exception message, ensure that you sanitize and validate it; for example, if you return an HTML error message, you should encode the output to avoid script injection.

  For more information, see Chapter 3, "Architecture and Design Guidelines."

- **How to instrument your application**
  Instrumentation is used when you have a specific problem and you need additional information in order to solve that problem. This could be a debugging issue, a performance issue, a security issue, a manageability issue, and so on. This is different than logging in that logging is a general approach to pushing information into log files that might need to be audited in the future, versus a targeted approach to get information for a specific problem. Options for instrumentation include event tracing for Microsoft® Windows®, trace and debug classes, custom performance counters, and Windows Management Instrumentation (WMI). For logging, consider the Logging Application Block in the Enterprise Library.

  For more information, see Chapter 3, "Architecture and Design Guidelines."

- **How to design for transactions**
  Use connection-based transactions when accessing a single data source. Where you cannot use transactions, implement compensating methods to revert the data store to its previous state. Avoid holding locks for long periods; for example, when using long-running atomic transactions. Consider using compensating locks for long-running transactions. If the chance of a data conflict from concurrent users is low (for example, when users are generally adding data or editing different rows), consider using optimistic locking during data access. If the chance of a data conflict from concurrent users is high (for example, when users are likely to be editing the same rows), consider using pessimistic locking during data access. If transactions take a long time to complete, consider using asynchronous transactions that call back to the client when complete. You can perform transactions using T-SQL commands, ADO.NET, or **System.Transaction**. T-SQL transactions are most efficient for server-controlled transactions on a single data store. Keep transactions as short as possible, consider your isolation level, and keep read operations to a minimum inside a transaction.

  For more information, see Chapter 3, "Architecture and Design Guidelines."

Presentation Layer

The following guidelines will help you to understand the fundamental factors you must consider when designing the presentation layer for your application. Use these guidelines as a starting point toward understanding key concerns, such as how to validate input on the client and on the server, and how to choose between the Model-View-Controller (MVC) and Model-View-Presenter (MVP) patterns.

- **How to validate input**
  Assume that all input data is malicious. Constrain, reject, and sanitize your input because it is easier to validate data for known valid types, patterns, and ranges than it is to validate data by looking for known bad characters. Validate data for type, length, format, and range. Consider using regular expressions for validating the inputs. For an improved user experience, consider using client-side validation, but always perform validation at the server as well. Encode your output.

For more information, see Chapter 3, “Architecture and Design Guidelines.”

- **How to use the MVC pattern**
  To use the Model-View-Controller (MVC) pattern effectively, you must understand the division of labor within the MVC triad (the Model, the View, and the Controller). You must also understand how the three parts of the triad communicate with each other to process requests from user input. The Model represents data, and the View is the UI, which displays the data to the user and contains controls for the user to interact with the data and the application. The Controller is responsible for handling requests, initializing the Model and choosing the appropriate View. Common Web implementations use HTTP handlers or Internet Server API (ISAPI) filters to intercept requests and send them directly to a controller. There are two main variations of the MVC pattern: the Passive Model and the Active Model. In the Passive Model pattern, changes to the Model are only captured when the Controller processes a request. However, in the Active Model pattern, an Observer pattern implementation can be used to notify the View and/or the Controller when changes occur in the Model. One of the limitations with this pattern is that, because the controller is responsible for intercepting and handling requests, you lose capabilities associated with the View such as the ability to handle control events and the use of viewstate in ASP.NET.

  For more information, see Chapter 10, “Presentation Layer Guidelines.”

- **How to use the MVP pattern**
  The Model-View-Presenter (MVP) pattern is very similar to the MVC pattern, with the main difference being that Views handle requests and pass them to a presenter, which provides controller logic. Similar to the controller in MVC, the Presenter is responsible for processing requests and initializing the model. The main advantage of using this pattern over MVC is that, because the view handles requests, you also have support for control events and the ability to maintain the state of controls in the view. There are two main variations on this pattern, Passive View and Supervising Controller. With Passive View, requests are intercepted by the View and passed to the Presenter, and then the Presenter initializes fields in the View through an interface. This variation completely decouples the view from the Model and provides the highest level of testability. With Supervising Controller, the View passes requests to the Presenter, the Presenter notifies the View when the Model is initialized, and the View accesses the Model directly. This variation is useful when you have a lot of data that needs to be presented in the View, which makes Passive View impractical.

  For more information, see Chapter 10, “Presentation Layer Guidelines.”

Business Layer

The following guidelines will help you to understand fundamental factors you must consider when designing the business layer for your application. Use these guidelines as a starting point toward understanding key concerns, such as when to use business entities, how to implement business components, and how to expose your business layer as a service.

- **How to implement business entities**
  If you are designing a small Web application or a service, and you want to take advantage of the disconnected behavior they provide, consider using **DataSets**. If you are working with content-based applications that have few or no business rules, consider using XML. If you have complex business rules related to the business domain, or if you are designing a rich client where the domain model can be initialized and held in memory, consider using custom Domain Model objects. If your tables or views in the database represent the business entities used by your application, consider using custom objects. If the data you are consuming is already in XML format, or if you are working with read-only document-based data, consider using custom XML objects.

  For more information, see Chapter 11, "Business Layer Guidelines."

- **How to implement business components**
  The type of business component you implement depends on your business rules. If you have volatile business rules, consider storing them in a separate rules engine. If you want your business rules to be separate from the business data, consider using business process components. If your business processes involve multiple steps with long-running transactions, consider using business workflow components. A well-designed business component exposes data and functionality based on how the data is used, and abstracts the underlying data store and service. Do not mix unrelated functionality within a business component; for example, do not mix data access logic and business logic within the same component. Consider designing consistent input and output data formats for business components.

  For more information, see Chapter 11, "Business Layer Guidelines."

- **How to design application service façades**
  A well-designed application service façade exposes a simple interface by encapsulating cohesive behavior that is specific to a set of related business operations. Avoid including business logic in a service interface in order to improve reusability and maintainability and reduce duplication of code. Consider using standard protocols such as the SOAP as the communication medium to ensure maximum compatibility with a range of clients.

  For more information, see Chapter 11, "Business Layer Guidelines."

Data Access Layer

The following guidelines will help you to understand the fundamental factors you need to consider when designing the data layer for your application. Use these guidelines as a starting point toward understanding the key concerns, such as how to design your data access layer, how to manage connections, the differences between stored procedures and dynamic SQL, how to think about data access performance, and how to effectively pass data through your application's layers and tiers.

- **How to design your data access layer**
  If you choose to access tables directly from your application without an intermediate data access layer, you may improve the performance of your application at the expense of maintainability. The data access logic layer provides a level of abstraction from the underlying data store. A well-designed data access layer exposes data and functionality based on how the data is used, and abstracts the underlying data store complexity. Do not arbitrarily map objects to tables and columns, and avoid deep object hierarchies. For example, if you want to display a subset of data, and your design retrieves an entire object graph instead of the necessary portions, there is unnecessary object creation overhead. Evaluate the data you need and how you want to use the data against your underlying data store.

  For more information, see Chapter 12, "Data Access Layer Guidelines."

- **How to design your connection-management approach**
  Pool connections. Connections are an expensive and scarce resource, which should be shared between callers by using connection pooling. Opening a connection for each caller limits scalability. Connect by using service accounts associated with a trusted subsystem security model. Open database connections only when you need them. Close the database connections as soon as you are finished—do not open them early, and do not hold them open across calls.

  For more information, see Chapter 12, "Data Access Layer Guidelines."

- **How to choose between stored procedures and dynamic SQL**
  When choosing between stored procedure and dynamic SQL, you need to consider the abstraction requirements, maintainability, and any environment constraints. Consider whether you need to implement abstraction in the database in the form of stored procedures or in the data layer in the form of dynamic SQL data access patterns or object/relational mapping (O/RM). If you have a small application with limited business rules, consider using dynamic SQL as it is the simplest model with the least development overhead. If you are building a larger application with maintainability requirements, consider using stored procedures since most changes to the database schema will have a minimal impact on application code. For security considerations, you should always use typed parameters, either passed to a stored procedure or used when creating dynamic SQL.

  For an in-depth discussion on how to make an informed choice, see Chapter 12, "Data Access Layer Guidelines."

- **How to improve data access performance**
  Minimize processing on the server and at the client. Minimize the amount of data passed over the network. Use database connection pooling to share connections across requests. Keep transactions as short as possible to minimize lock durations and to improve concurrency. However, do not make transactions so short that access to the database becomes too chatty.

  For more information, see Chapter 12, "Data Access Layer Guidelines."

- **How to pass data across layers and tiers**
  Consider scalar values when the consumer is interested only in the data and not the type or structure of the entity. Do not consider scalar values if your design is not capable of handing schema changes. Consider XML strings when you must support a variety of callers, including third-party clients. Consider custom objects when you must handle complex data, communicate with components that know about the object type, or require better performance through binary serialization. Consider using Data Transfer Objects (DTOs) that combine multiple data structures into a single structure in order to reduce round trips between layers and tiers. With the Microsoft .NET Framework, do not pass **DataReader** objects between layers because they require an open connection. Consider using **DataSets** for disconnected scenarios in simple CRUD (Create, Read, Update, Delete)-based applications. A **DataSet** contains schema information and maintains a change record, which means it can be modified and used to update the database without requiring additional code. However, it is important to understand that **DataSets** are expensive to create and serialize compared to custom objects.

  For more information, see Chapter 12, "Data Access Layer Guidelines."

Services

The following guidelines will help you to understand the fundamental factors you need to consider when designing the service layer for your application. Use these guidelines as a starting point toward understanding the key concerns, such as when to use business entities, how to design your service interface, how to choose a Web service technology, and how to think about Representational State Transfer (REST) and SOAP.

- **How to design a service**
  When designing services, you must consider the availability and stability of the service, and ensure that it is configurable and can be aggregated so that it can accommodate changes to the business requirements. In most cases, you want to design services that are autonomous, provide explicit boundaries, do not expose internal classes, and use policy to define interaction requirements. You should also design for idempotency so that the service can manage messages that arrive more than once; design for commutativity so that the service can handle messages that arrive in the wrong order; and design for invalid requests

by validating them against a schema or known format. Consider using standard elements to compose the complex types used by your service.

For more information, see Chapter 13, "Service Layer Guidelines."

- **How to expose your application as a service**
  The approach you take to exposing an application as a service depends on where you are in the development life cycle of the application. If you already have an application and want to expose operations as services, you should identify the necessary operations and then define interface contracts that combine operations in order to produce application-scoped procedures. The main thing you do not want to do is expose component- or object-based operations as service operations. When starting from scratch, you should first define the service interface contracts that an application needs to support. The main goal is to provide coarse-grained interface contracts that support business-process or client requirements without focusing on the implementation. Once you have defined the contracts, you can then focus on how to implement code that supports the contracts.

  For more information, see Chapter 13, "Service Layer Guidelines."

- **How to choose between ASP.NET Web services and WCF services for services**
  ASP.NET Web services are a good choice for simple HTTP-based services hosted in Internet Information Services (IIS). WCF is a good choice if you need the performance of Transmission Control Protocol (TCP) communication over HTTP, or if you need to host the service without a Web server. WCF provides support for the WS* specification, which includes support for end-to-end security and reliable communication. WCF allows you to implement duplex communication, and you can also use it with Windows Message Queuing and as a Windows service. In addition, you have more options with regard to protocols, bindings, and formats. Keep in mind that WCF requires the .NET Framework 3.0 or higher.

  For more information, see Chapter 13, "Service Layer Guidelines."

- **How to choose between REST and SOAP**
  Representational State Transfer (REST) and SOAP represent two different styles for implementing services. REST uses HTTP, which means that it works very much like a Web application, while SOAP is an XML-based messaging protocol that can be used with any communication protocol. Although both REST and SOAP can be used with most service implementations, the REST architectural style is better suited for public services or cases where a service can be accessed by unknown consumers. SOAP is better suited for implementing a loosely coupled Remote Procedure Call (RPC) interface between layers of an application. With SOAP, you are not restricted to HTTP, and it provides the underlying framework for more advanced Web Service standards, such as the ability to enlist in transactions.

  For more information, see Chapter 13, "Service Layer Guidelines."

Quality Attributes

The following guidelines will help you to understand how focusing on the quality attributes can produce a more successful design. The guidelines help you to understand how to design your application, keeping security and performance in mind right from the beginning.

- **How to design for security**
  Use threat modeling to systematically identify threats instead of applying security in a haphazard manner. Itemize your application's important characteristics, assets, and actors to help you identify threats. A detailed understanding of your application will also help you uncover more relevant and detailed threats. Use a security frame to focus on areas where mistakes are most often made. Key categories in a security frame include auditing and logging, authentication, authorization, configuration management, cryptography, exception management, input and data validation, and sensitive data. Rate the threats based on the risk of an attack or occurrence of a security compromise and the potential damage that could result. This allows you to deal with threats in the appropriate order.

  For more information, see Chapter 7, "Quality Attributes."

- **How to design for performance**
  Use performance modeling early in the design process to help you evaluate your design decisions against your objectives before you commit time and resources. Identify your performance objectives, your workload, and your budgets. For example, performance objectives may include maximum execution time and resource utilization such as CPU, memory, disk I/O, and network I/O. Identify your constraints, such as time and hardware budget. Use load testing and unit tests to measure performance, and identify if hardware or deployment scenarios are the cause of bottlenecks. Ensure that you test with data types and data volumes that match the actual run-time scenarios.

  For more information, see Chapter 7, "Quality Attributes."

- **How to identify and evaluate performance issues**
  Focus on the critical areas where the correct approach is essential and where mistakes are often made. Identify requirements, cost, and budget restraints, and whether improvements can come from additional hardware and infrastructure, improved application code, or by adopting a different deployment approach. Perform design inspections and code inspections to identify poor practices that could lead to bottlenecks. Organize and prioritize your performance issues by using a performance frame. Key categories in a performance frame include data structures and algorithms, communication, concurrency, resource management, coupling and cohesion, caching, and state management.

  For more information, see Chapter 7, "Quality Attributes."

Fast Track: A Guide for Getting Started and Applying the Guidance

Objectives

- Understand the key components of this guide.
- Learn about the canonical layered application style.
- Learn the steps you should follow when beginning your design.
- Learn about the main application types and architectural styles.
- Learn the quality attributes and understand the engineering hotspots that are important when designing an application.

Overview

This “fast track” chapter highlights the basic approach taken by this guide to help you design and architect layered applications across a variety of application types and architecture styles. Use this chapter to understand the basic approach, application types, architecture styles, the quality attributes that impact application design, and the key engineering decisions to consider when designing application architecture.

Architecture Meta-frame

The architecture meta-frame can help you to focus on the key factors that can influence your design. Use the meta-frame to formulate how you will design your architecture, to help you ask key questions when reviewing your architecture, and as a way to organize your thoughts during design activities. This guide is organized around the application types, architecture styles, and architecture frame described by this architecture meta-frame.

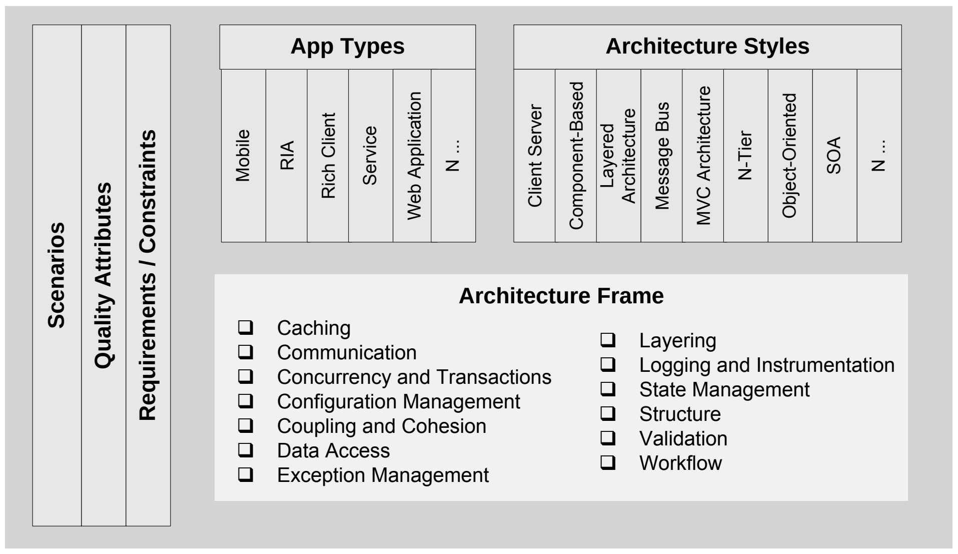

Figure 1 The architecture meta-frame

The meta-frame contains the following key components:

- **Scenarios**. Application scenarios tie architecture solutions to the real-world scenarios that impact your application design. For example, your application may map to an Internet Web application scenario, which has unique architecture solutions compared to a mobile client application.
- **Quality attributes**. Quality attributes represent cross-cutting concerns that apply across application types, and should be considered regardless of architecture style. Security, performance, maintainability, and reusability are examples of quality attributes.
- **Requirements and constraints**. Requirements and constraints narrow the range of possible solutions for your application architecture problems.
- **Application types**. Application types categorize the major application technology stacks on the Microsoft platform. Examples of application types include mobile, rich Internet application (RIA), services application, and Web application.
- **Architecture styles**. An architectural style is a collection of principles that shapes the design of your application. Many of these styles overlap and can be used in combination. Architectural styles tend to be tied both to the application type and to the point in time in which the application was developed.
- **Architecture frame.** The architecture frame is a collection of hotspots that you can use to analyze your application architecture. This helps you to turn core features such as caching, data access, validation, and workflow into actions.

Reference Application Architecture

The reference application architecture represents a canonical view of a typical application architecture, using a layered style to separate functional areas into separate layers. The reference application architecture demonstrates how a typical application might interact with its users, external systems, data sources, and services. The reference application architecture also shows how cross-cutting concerns such as security and communication impact all of the layers in your design, and must be designed with the entire application in mind.

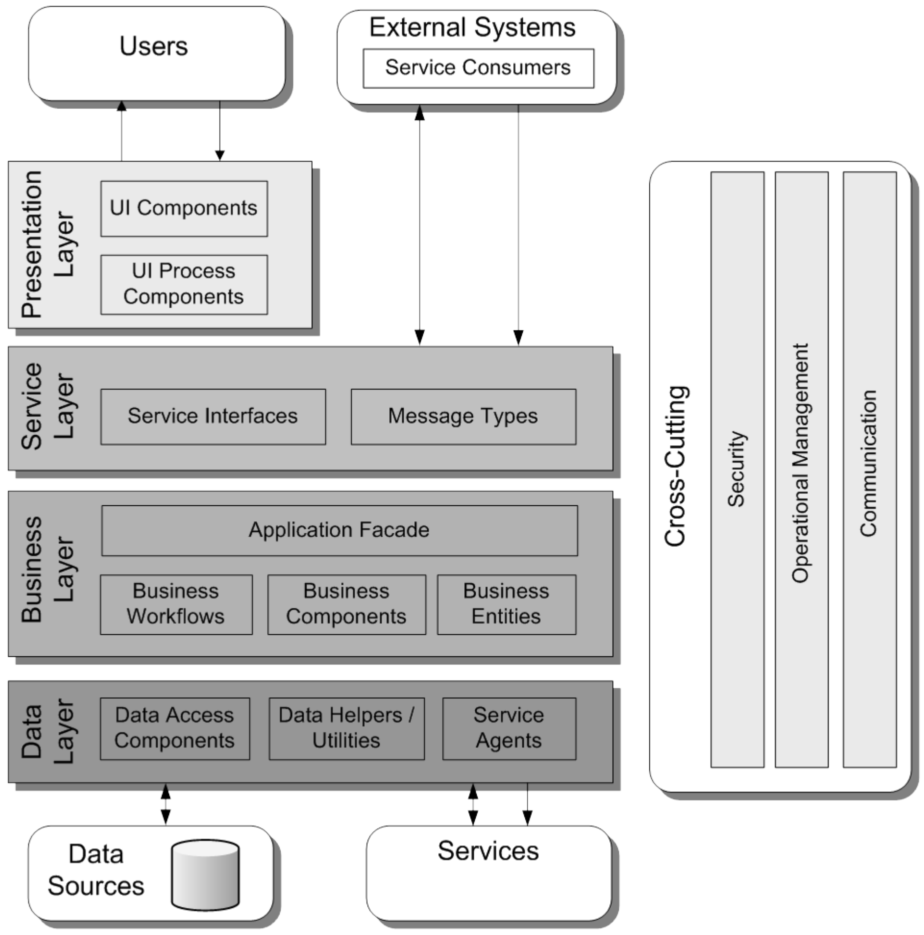

Figure 2 The reference application architecture

Presentation Layer Components

- **User interface (UI) components**. UI components provide a way for users to interact with the application. They render and format data for users and acquire and validate data input by the user.
- **User process components**. To help synchronize and orchestrate these user interactions, it can be useful to drive the process by using separate user process components. This means that the process-flow and state-management logic is not hard-coded in the UI elements themselves, and the same basic user interaction patterns can be reused by multiple UIs.

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.

Business Layer Components

- **Application facade** (optional). Use a façade to combine multiple business operations into a single message-based operation. You might access the application façade from the presentation layer by using different communication technologies.
- **Business components**. 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 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 used internally in the application are usually data structures, such as **DataSets**, **DataReaders**, or Extensible Markup Language (XML) streams, but they can also be implemented by 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 workflows**. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. Business workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.

Data Layer Components

- **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 the process easier to configure and maintain.
- **Data helpers / utility components**. 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**. Service agents isolate your application from 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.

Designing Your Architecture

The approach to architectural design can be divided into the following steps:

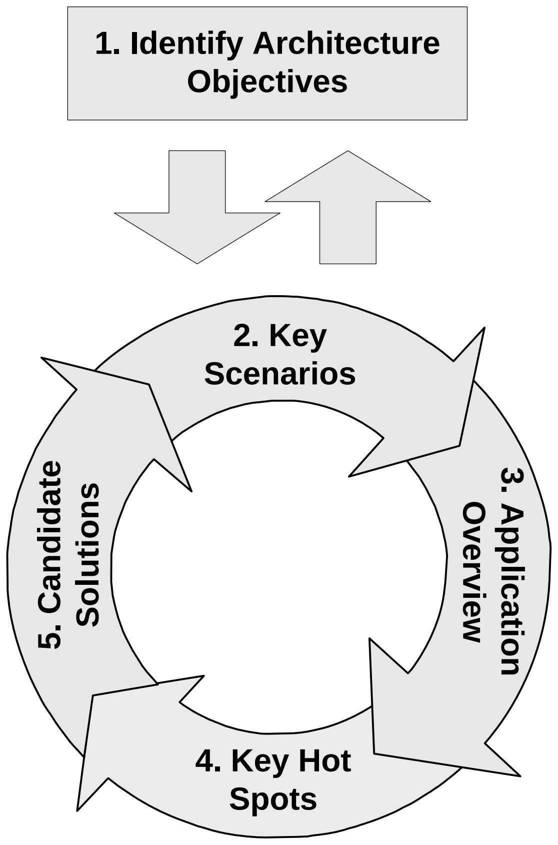

Figure 3 Core architecture design activities

These steps are:

- **Step 1: Identify Architecture Objectives**. Clear objectives help you to focus on your architecture, and on solving the right problems in your design. Good objectives help you to determine when you are finished with the current phase, and when you are ready to move on to the next phase.
- **Step 2**: **Key Scenarios.** Use key scenarios to focus your design on what matters most, and to evaluate your candidate architectures when they are ready.
- **Step 3: Application Overview.** Understand your application type, deployment architecture, architectural styles, and technologies in order to connect your design to the real world in which the application will have to operate.
- **Step 4: Key Hotspots.** Identify key hotspots based on quality attributes and the architecture frame. These are the areas where mistakes are most often made when designing an application.

- **Step 5: Candidate Solutions.** Create a candidate architecture or architectural spike and evaluate it against your key scenarios, hotspots, and deployment constraints.

This approach will allow you to design a candidate architecture that can be reviewed, tested, and compared to your requirements and constraints. You can iteratively flesh out your architecture as you work through your design and discover more details that impact your architecture. You do not have to build your architecture in a single iteration. Don't get lost in the details; focus on the big steps and build a framework on which you can base your architecture and design.

Application Types

Your choice of application type will be related both to the technology constraints and the type of user experience you plan to deliver. Use scenarios to help you choose an application type. For example, if you want to support rich media and graphics delivered over the Internet, a rich Internet application (RIA) is probably the best choice. However, if you want to support data entry with forms in an occasionally connected scenario, a rich client is probably the best choice. Use the table below to review and understand each application type.

Table 1 Application types

Application type	Description
Mobile Application	• Can be developed as a Web application or a rich client application. • Can support occasionally connected scenarios. • Runs on devices with limited hardware resources.
Rich Client Application	• Usually developed as a stand-alone application. • Can support disconnected or occasionally connected scenarios. • Uses the processing and storage resources of the local machine.
Rich Internet Application	• Can support multiple platforms and browsers. • Can be deployed over the Internet. • Designed for rich media and graphical content. • Runs in the browser sandbox for maximum security. • Can use the processing and storage resources of the local machine.
Service Application	• Designed to support loose coupling between distributed components. • Service operations are called using XML-based messages. • Can be accessed from the local machine or remotely, depending on the transport protocol.
Web Application	• Can support multiple platforms and browsers. • Supports only connected scenarios. • Uses the processing and storage resources of the server.

Architectural Styles

Your choice of architectural styles will depend upon your application type, the requirements and constraints of your application, the scenarios you want to support, and—to some extent—

the styles with which you are most familiar and comfortable. Your choice of architectural styles represents a set of principles that your design will follow—an organizing set of ideas that you can use to keep your design cohesive and focused on your key objectives and scenarios. Use the table below to review and understand the key set of architectural styles.

Table 2 Architectural styles

Architecture style	Description
Client-Server	Segregates the system into two computer programs where one program, the client, makes a service request to another program, the server.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that are location-transparent and expose well-defined communication interfaces.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message-Bus	A software system that can receive and send messages that are based on a set of known formats, so that systems can communicate with each other without needing to know the actual recipient.
Model-View-Controller (MVC)	Separates the logic for managing user interaction from the UI view and from the data with which the user works.
N-tier / 3-tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	An architectural style based on division of tasks for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.

Quality Attributes

Use the quality attributes to focus your thinking around critical problems that your design should solve. Addressing quality attributes in your design rather than during development will improve the likelihood that your application will be successful in the long term. Use the table below to review and understand each quality attribute.

Table 3 Quality attributes

Category	Description
Availability	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
Conceptual Integrity	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.

Category	Description
Flexibility	Flexibility is the ability of a system to adapt to varying environments and situations, and to cope with changes to business policies and rules. A flexible system is one that is easy to reconfigure or adapt in response to different user and system requirements.
Interoperability	Interoperability is the ability of diverse components of a system or different systems to operate successfully by exchanging information, often by using services. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
Maintainability	Maintainability is the ability of a system to undergo changes to its components, services, features, and interfaces as may be required when adding or changing the functionality, fixing errors, and meeting new business requirements.
Manageability	Manageability defines how easy it is to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
Performance	Performance is an indication of the responsiveness of a system to execute any action within a given interval of time. It can be measured in terms of latency or throughput. *Latency* is the time taken to respond to any event. *Throughput* is the number of events that take place within given amount of time.
Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified interval of time.
Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Scalability	Scalability is the ability of a system to function well when there are changes to the load or demand. Typically, the system will be able to be extended by scaling up the performance of the server, or by scaling out to multiple servers as demand and load increase.
Security	Security defines the ways that a system is protected from disclosure or loss of information, and the possibility of a successful malicious attack. A secure system aims to protect assets and prevent unauthorized modification of information.
Supportability	Supportability defines how easy it is for operators, developers, and users to understand and use the application, and how easy it is to resolve errors when the system fails to work correctly.
Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.

Category	Description
Usability	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, able to provide good access for disabled users, and able to provide a good overall user experience.

Architecture Frame

The architecture frame is a collection of hotspots that represent key engineering decisions. Each represents an opportunity to improve your design and build a technically more effective architecture. This architecture frame is part of the larger architecture meta-frame, and is used throughout the guide to organize key patterns, principles and practices. These categories help you to focus on the most important areas, and obtain the most meaningful and actionable guidance.

Categories

Use the table below to review and understand each category in the architecture frame.

Table 4 Architecture frame categories

Category	Description
Authentication and Authorization	Authentication and authorization allow you to identify the users of your application with confidence, and to determine the resources and operations to which they should have access.
Caching and State	Caching improves performance, reduces server round trips, and can be used to maintain the state of your application.
Communication	Communication strategies determine how you will communicate between layers and tiers, including protocol, security, and communication-style decisions.
Composition	Composition strategies determine how you manage component dependencies and the interactions between components.
Concurrency and Transactions	Concurrency is concerned with the way that your application handles conflicts caused by multiple users creating, reading, updating, and deleting data at the same time. Transactions are used for important multi-step operations in order to treat them as though they were atomic, and to recover in the case of a failure or error.
Configuration Management	Configuration management defines how you configure your application after deployment, where you store configuration data, and how you protect the configuration data.
Coupling and Cohesion	Coupling and cohesion are strategies concerned with layering, separating application components and layers, and organizing your application trust and functionality boundaries.

Category	Description
Data Access	Data access strategies describe techniques for abstracting and accessing data in your data store. This includes data entity design, error management, and managing database connections.
Exception Management	Exception-management strategies describe techniques for handling errors, logging errors for auditing purposes, and notifying users of error conditions.
Logging and Instrumentation	Logging and instrumentation represents the strategies for logging key business events, security actions, and provision of an audit trail in the case of an attack or failure.
User Experience	User experience is the interaction between your users and your application. A good user experience can improve the efficiency and effectiveness of the application, while a poor user experience may deter users from using an otherwise well-designed application.
Validation	Validation is the means by which your application checks and verifies input from all sources before trusting and processing it. A good input and data-validation strategy takes into account not only the source of the data, but also how the data will be used, when determining how to validate it.
Workflow	Workflow is a system-assisted process that is divided into a series of execution steps, events, and conditions. The workflow may be an orchestration between a set of components and systems, or it may include human collaboration.

Key Engineering Decisions

Use the architecture frame as a way to organize and think about key engineering decisions. The following table shows key engineering decisions for each category in the architecture frame.

Table 5 Key engineering decisions

Category	Key problems
Authentication and Authorization	• How to store user identities • How to authenticate callers • How to authorize callers • How to flow identity across layers and tiers

Category	Key problems
Caching and State	• How to choose effective caching strategies • How to improve performance by using caching • How to improve availability by using caching • How to keep cached data up to date • How to determine the data to cache • How to determine where to cache the data • How to determine an expiration policy and scavenging mechanism • How to load the cache data • How to synchronize caches across a Web or application farm
Communication	• How to communicate between layers and tiers • How to perform asynchronous communication • How to communicate sensitive data
Composition	• How to design for composition • How to design loose coupling between modules • How to handle dependencies in a loosely coupled way
Concurrency and Transactions	• How to handle concurrency between threads • How to choose between optimistic and pessimistic concurrency • How to handle distributed transactions • How to handle long-running transactions • How to determine appropriate transaction isolation levels • How to determine whether compensating transactions are required
Configuration Management	• How to determine the information that must be configurable • How to determine location and techniques for storing configuration information • How to handle sensitive configuration information • How to handle configuration information in a farm or cluster
Coupling and Cohesion	• How to separate concerns • How to structure the application • How to choose an appropriate layering strategy • How to establish boundaries
Data Access	• How to manage database connections • How to handle exceptions • How to improve performance • How to improve manageability • How to handle binary large objects (BLOBs) • How to page records • How to perform transactions
Exception Management	• How to handle exceptions • How to log exceptions

Category	**Key problems**
Logging and Instrumentation	• How to determine the information to log • How to make logging configurable
User Experience	• How to improve task efficiency and effectiveness • How to improve responsiveness • How to improve user empowerment • How to improve the look and feel
Validation	• How to determine location and techniques for validation • How to validate for length, range, format, and type • How to constrain and reject input • How to sanitize output
Workflow	• How to handle concurrency issues within a workflow • How to handle task failure within a workflow • How to orchestrate processes within a workflow

PART I

Fundamentals

In This Part:

- ▶ **Fundamentals of Application Architecture**
- ▶ **.NET Platform Overview**
- ▶ **Architecture and Design Guidelines**

Chapter 1: Fundamentals of Application Architecture

Objectives

- Learn the fundamental concepts of application architecture.
- Understand key application architecture terms and principles.
- Learn about the key forces shaping today's architectural landscape.

Overview

Application architecture is the process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security, and manageability. It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application. By following the processes described in this guide, and using the information it contains, you will be able to construct architectural solutions that address all of the relevant concerns, can be deployed on your chosen infrastructure, and provide results that meet the original aims and objectives.

This chapter lays a foundation for practical application architecture. It starts by describing architecture and design at a high level and then dives deeper into specific aspects of application architecture and design. The remainder of the guide follows the same approach. Finally, this chapter defines key terms and principles related to application architecture. Understanding these will help you gain the most benefit from this guide and be more successful as an application architect. After you work through his chapter, you will understand the overall techniques and goals of application architecture and design, the major factors you must consider, and the different ways that you can approach application design. This will help you to understand concepts such as architectural and application styles, quality attributes and cross-cutting concerns, and analyzing and representing architecture.

What Is Application Architecture?

Software architecture is often defined as the structure or structures of a system. Several well-known industry experts have expanded that definition with information on the decisions that must be made related to architecture. Here we look at a few somewhat formal definitions of architecture, and then take a more informal view.

Kruchten, Booch, Bittner, and Reitman on Architecture

Philippe Kruchten, Grady Booch, Kurt Bittner, and Rich Reitman derived and refined a definition of architecture based on work by Mary Shaw and David Garlan (Shaw and Garlan 1996). Their definition is:

> "Software architecture encompasses the set of significant decisions about the organization of a software system including:
>
> Selection of the structural elements and their interfaces by which the system is composed.
> Behavior as specified in collaboration among those elements.
> Composition of these structural and behavioral elements into larger subsystems.
> Architectural style that guides this organization.
>
> Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns."

Fowler on Architecture

In *Patterns of Enterprise Application Architecture*, Martin Fowler outlines some common recurring themes when explaining architecture:

- The highest-level breakdown of a system into its parts.
- The decisions that are hard to change.
- There are multiple architectures in a system.
- What is architecturally significant can change over a system's lifetime.
- In the end, architecture boils down to whatever the important stuff is.

Bass, Clements, and Kazman on Architecture

In *Software Architecture in Practice (2nd edition)*, Bass, Clements, and Kazman define architecture as follows:

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural."

Why Do We Need Architecture?

Like any other complex structure, software must be built on a solid foundation. Failing to consider key scenarios, failing to design for common problems, or failing to appreciate the long-term consequences of key decisions can put your application at risk. Modern tools and platforms help to simplify the tasks of building applications, but they do not replace the requirement to design your application based on your specific scenarios. The risks exposed by poor architecture include software that is unstable, is unable to support business requirements, or could even prevent the application from working when deployed into a production environment.

Consider the following high-level concerns when thinking about software architecture:

- How will the application be deployed into production?

- How will the users be using the application?
- What are the quality attribute requirements, such as security, performance, concurrency, internationalization, and configuration?
- What are the architectural trends that might impact your application now or after it has been deployed?

Architecture vs. Design

According to Martin Fowler in the "Who Needs an Architect?" available at http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01231144:

> "…the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.' This understanding includes how the system is divided into components and how the components interact through interfaces. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers."

Therefore architecture focuses on how components and interfaces are used by or interact with other components. Selecting data structures or algorithms implemented within the components is not an architectural concern. Rather than use hard-and-fast rules to distinguish between architecture and design, it makes sense to combine these two areas. In some cases, decisions are clearly more architectural in nature. In other cases, the decisions are more about design, and how they help you to realize that architecture.

User, Business, and System Goals

Systems should be architected with consideration for the user, the system, and the business goals. For each of these areas, outline key scenarios, important quality attributes (for example, maintainability), key satisfiers, and key dissatisfiers. When possible, develop and consider metrics that measure success in each of these areas.

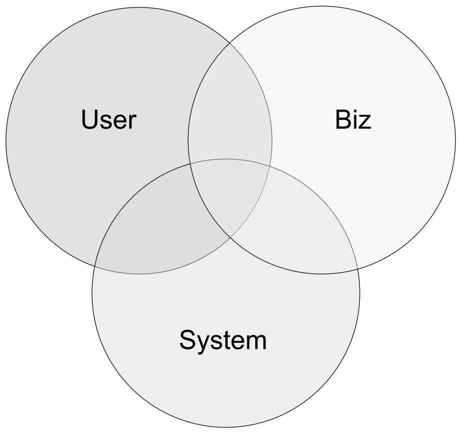

Figure 1 User, business, and system goals

Tradeoffs are likely between each area, and a balance point must be found. For example, responsiveness might be a major user goal, but the system administrator is not willing to invest in the hardware required to meet that goal for 100 percent of the time. A balance point might be to meet the goal only 80 percent of the time.

The Goals of Architecture

Application architecture seeks to build a bridge between business requirements and technical requirements by understanding use cases, and then finding ways to implement those use cases in the software. The goal of architecture is to identify the requirements that impact the structure of the application. Good architecture reduces the business risks associated with building a technical solution. A good design is sufficiently flexible to be able to handle the natural drift that will occur over time in hardware and software technology, as well as in user scenarios and requirements. An architect must consider the overall impact of design decisions, the inherent tradeoffs between quality attributes (such as performance and security), and the tradeoffs required to address user, system, and business requirements.

Keep in mind that the architecture should:

- Expose the structure of the system but hide the implementation details.
- Realize all of the use-case scenarios.
- Try to address the concerns of various stakeholders.
- Handle both functional and quality requirements.

Approach to Architecture

There are key decisions that must be addressed with any architecture, regardless of your architectural approach. At a minimum, you must determine the type of application that you are building, the architectural styles that will be used, and how you will handle cross-cutting concerns. Throughout this guide, we use an architectural baseline for framing out the different areas that must be addressed in your architecture. The architectural baseline is shown in the following diagram.

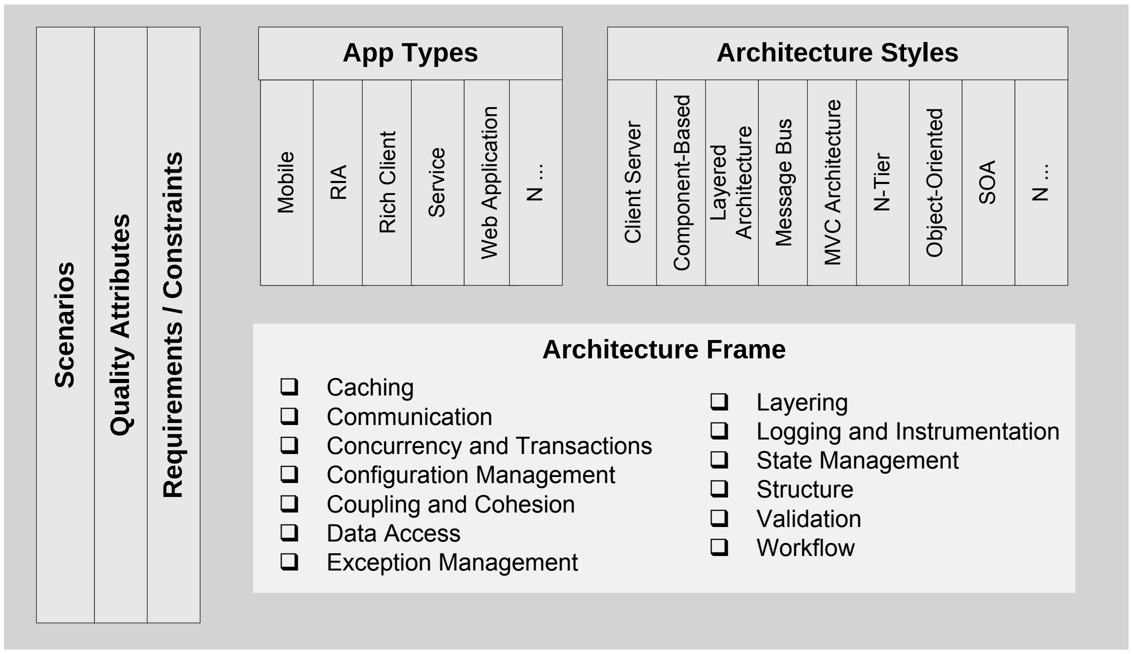

Figure 2 The architecture meta-frame

In addition to the architectural baseline, you can use the following approach to help define your architecture. The first step is to identify the type of application you plan to build. Next, you must understand how the application will be deployed. Once you know what type of application you are building and how it will be deployed, you can start to drill down into the architecture to identify styles and technologies that you will use. Finally, you must consider how quality attributes and cross-cutting concerns will be incorporated into the design of your system.

Application Type

Choosing the right application type is the key part of the process of designing and architecting an application. Your choice of an appropriate application type is governed by your specific requirements and infrastructure limitations. This guide covers the following application types:

- Mobile applications designed for mobile devices.
- Rich client applications designed to run primarily on a client PC.
- Rich Internet applications designed to be deployed from the Internet, which support rich user interface (UI) and media scenarios.
- Service applications designed to support communication between loosely coupled components.
- Web applications designed to run primarily on the server in fully connected scenarios.

Deployment Strategy

When you design your application architecture, you must take into account corporate policies and procedures, together with the infrastructure on which you plan to deploy your application. Whether or not the target environment is inflexible, your application design must accommodate any restrictions that exist in that environment. Your application design must also take into account quality attributes such as security, performance, and maintainability. Sometimes you must make design tradeoffs due to protocol restrictions and network topologies.

Identify the requirements and constraints that exist between the application architecture and infrastructure architecture early in the design process. This helps you to choose an appropriate deployment topology, and to resolve conflicts between the application and infrastructure architecture early in the process.

Architectural Style

An architectural style is a set of principles. Each style defines a set of rules that specify the kinds of components you can use to assemble a system, the kinds of relationships used in their assembly, constraints on the way they are assembled, and assumptions about the meaning of how you put them together. Examples of architectural styles are client/server, component-based, layered architecture, message-bus, Model-View-Controller (MVC), 3-tier/N-tier, object-oriented, and service-oriented architecture (SOA). Many factors can influence your choice of architectural styles. These include the capacity of your organization for design and implementation, the capabilities and experience of developers, and the infrastructure constraints and deployment scenarios available.

Appropriate Technologies

When choosing technologies for your application, the key factors to consider are the type of application you are developing and your preferred options for application deployment topology and architectural styles. Your choice of technologies will also be governed by organization policies, infrastructure limitations, resource skills, and so on. You must compare the capabilities of the technologies you choose against your application requirements, taking into account all of these factors before making decisions

Quality Attributes

Quality attributes—such as security, performance, and maintainability—can be used to focus your thinking around the critical problems that your design should solve. Depending on your requirements, you might need to consider every quality attribute covered in this guide, or you might only need to consider a subset. For example, every application design must consider security and performance, but not every design needs to consider interoperability or scalability. Understand your requirements and deployment scenarios first so that you know which quality attributes are important for your design. Keep in mind that quality attributes may conflict; for example, security often requires a tradeoff against performance or usability. Analyze and understand the key tradeoffs when designing for security attributes so that side effects do not become obvious later.

When designing to accommodate quality attributes, consider the following guidelines:

- Quality attributes are system properties that are separate from the functionality of the system.
- From a technical perspective, implementing quality attributes can differentiate a good system from a bad one.
- There are two types of quality attributes: those that are measured at run time, and those that can only be estimated through inspection.
- Analyze the tradeoffs between quality attributes.

Questions you should ask when considering quality attributes include:
- What are the key quality attributes required for your application? Identify them as part of the design process.
- What are the key requirements for addressing these attributes? Are they actually quantifiable?
- What are the acceptance criteria that will indicate that you have met the requirements?

Cross-Cutting Concerns

Cross-cutting concerns represent key areas of your design that are not related to a specific layer in your application. For example, you might want to cache data in the presentation layer, the business layer, and the data access layer. In such a case you will also need to design an exception-management framework that will work within each layer and across the layers as exceptions are propagated through the system. In addition, you should design logging so that each layer can log to a common store and the results can be correlated, and you should design a communication approach that you can use to communicate between the layers. Because authentication and authorization concerns also exist across multiple layers, you must determine how to pass identities and grant access to resources through the system.

The following list describes the key cross-cutting concerns that you must consider when architecting your applications:
- **Authentication**. Determine how to authenticate your users and pass authenticated identities across the layers.
- **Authorization**. Ensure proper authorization with appropriate granularity within each layer, and across trust boundaries.
- **Caching**. Identify what should be cached, and where to cache, to improve your application’s performance and responsiveness. Ensure that you consider Web farm and application farm issues when designing caching.
- **Communication**. Choose appropriate protocols, reduce calls across the network, and protect sensitive data passing over the network.
- **Exception management**. Catch exceptions at the boundaries. Do not reveal sensitive information to end users.
- **Instrumentation and logging**. Instrument all of the business- and system-critical events, and log sufficient details to recreate events in your system. Do not log sensitive information.

Designing Your Architecture

Modern thinking on architecture assumes that your design will evolve over time and that you cannot know everything you need to know up front in order to fully architect your system. Your design will generally need to evolve during the implementation stages of the application as you learn more, and as you test the design against real-world requirements. Create your architecture with this evolution in mind so that it will be agile in terms of adapting to requirements that are not fully known at the start of the design process.

Consider the following questions as you create an architectural design with agility in mind:

- What are the foundational parts of the architecture that represent the greatest risk if you get them wrong?
- What are the parts of the architecture that are most likely to change, or whose design you can delay until later with little impact?
- What are your key assumptions, and how will you test them?
- What conditions may require you to refactor the design?

Do not attempt to over-engineer the architecture, and do not make assumptions that you cannot verify. Instead, keep your options open for future change, and do not paint yourself into a corner. There will be aspects of your design that you must fix early in the process, which may represent significant cost if redesign is required. Identify these areas quickly and invest the time necessary to get them right.

Key Architecture Principles

Consider the following key principles when designing your architecture:

- **Build to change over build to last**. Wherever possible, design your application so that it can change over time to address new requirements and challenges.
- **Model to analyze and reduce risk**. Use threat models to understand risks and vulnerabilities. Use design tools and modeling systems such as Unified Modeling Language (UML) where appropriate.
- **Models and views are a communication and collaboration tool**. Efficient communication of design principles and design changes is critical to good architecture. Use models and other visualizations to communicate your design efficiently and to enable rapid communication of changes to the design.
- **Identify key engineering decisions**. Use the architecture frame in this guide to understand the key engineering decisions and the areas where mistakes are most often made. Invest in getting these key decisions right the first time so that the design is more flexible and less likely to be broken by changes.

Incremental and Iterative Approach to Architecture

Consider using an incremental and iterative approach to refining your architecture. Do not try to get it all right the first time—design just as much as you can in order to start testing the design against requirements and assumptions. Iteratively add details to the design over multiple passes to make sure that you get the big decisions right first, and then focus on the details. A common pitfall is to dive into the details too quickly and get the big decisions wrong by making incorrect assumptions, or by failing to evaluate your architecture effectively.

Use baseline architectures to get the big picture right, and use candidate architectures to iteratively test and improve your architecture. When testing your architecture, consider the following questions:

- What assumptions have I made in this architecture?
- What explicit or implied requirements is this architecture meeting?
- What are the key risks with this architectural approach?
- What countermeasures are in place to mitigate key risks?
- In what ways is this architecture an improvement over the baseline or the last candidate architecture?

Baseline and Candidate Architectures

A *baseline architecture* describes the existing system—it is how your system looks today. If this is a new architecture, your initial baseline is the first high-level architectural design from which candidate architectures will be built. A candidate architecture includes the application type, the deployment architecture, the architectural style, technology choices, quality attributes, and cross-cutting concerns.

Architectural Spikes

An *architectural spike* is an end-to-end test of a small segment of the application. The purpose of an architectural spike is to reduce risk and to test potential paths. As you evolve your architecture, you may use spikes to explore different scenarios without impacting the existing design. An architectural spike will result in a candidate architecture that can be tested against a baseline. If the candidate architecture is an improvement, it can become the new baseline from which new candidate architectures can be created and tested. This iterative and incremental approach allows you to get the big risks out of the way first, iteratively render your architecture, and use architectural tests to prove that each new baseline is an improvement over the last.

Consider the following questions to help you test a new candidate architecture that results from an architectural spike:

- Does this architecture introduce new risks?
- Does this architecture mitigate additional known risks?
- Does this architecture meet additional requirements?
- Does this architecture enable architecturally significant use cases?
- Does this architecture address quality attribute concerns?
- Does this architecture address additional cross-cutting concerns?

Architecturally Significant Use Cases

Architecturally significant use cases are those that meet the following criteria:

- They are important for the success and acceptance of the deployed application.

- They exercise enough of the design to be useful in evaluating the architecture.

After you have determined architecturally significant use cases for your application, you can use them as a way to evaluate the success or failure of candidate architectures. If the candidate architecture addresses more use cases, or addresses existing use cases more effectively, it will help you to determine that this candidate architecture is an improvement over the baseline architecture.

Analyzing and Evaluating Architecture

Use architecture evaluation to determine the feasibility of your baseline and candidate architectures. Architecture evaluation is a key component of successful architecture iterations. Consider the following techniques for architecture evaluation:

- **Architecturally significant use cases**. Test your design against use cases that are important to the success of your application, and which exercise a significant portion of the design.
- **Scenario-based evaluations**. Use scenarios to analyze your design with a focus on quality attributes. Examples of scenario-based evaluations are: Architecture Trade-off Analysis Method (ATAM), Software Architecture Analysis Method (SAAM), and Active Reviews for Intermediate Designs (ARID).

Representing and Communicating the Architecture

Communicating your design is critical for architecture reviews, as well as for the use of your architecture during implementation. In the end, your architecture is only as good as the quality of communication of your ideas. You must communicate your architectural design to a variety of roles, including system designers, developers, system administrators, and management.

One way to think of an architectural view is as a map of the important decisions. The map is not the terrain; instead, it is an abstraction that helps you to share and communicate the architecture.

The Architectural Landscape

Understand the key forces that are shaping architectural decisions today, and which will change how architectural decisions are made in the future. These key forces are driven by user demand, as well as by business demand for faster results, better support for varying work styles and workflows, and improved adaptability of software design.

Consider the following key trends:

- **User empowerment**. A design that supports user empowerment is flexible, configurable, and focused on the user experience. Design your application with user personalization and options in mind. Allow the user to define how they interact with your application instead of dictating to them. Understand the key scenarios and make them as simple as possible; make it easy to find information and use the application.
- **Market maturity**. Take advantage of market maturity by leveraging existing platform and technology options. Build on higher-level application frameworks where it makes sense to,

so that you can focus on what is uniquely valuable in your application, rather than building what already exists and can be reused. Leverage patterns that provide rich sources of proven solutions for common problems.

- **Agility and adaptability**. An agile, adaptable design takes advantage of loose coupling to allow reuse. Take advantage of pluggable designs to provide extensibility. Take advantage of service-orientation techniques such as SOA to provide interoperability.
- **Future trends**. When building your architecture, understand the future trends that might impact your design after deployment. For example, consider trends in rich UI and media, composition models such as mashups, increasing network bandwidth and availability, increasing use of mobile devices, continued improvement of hardware performance, interest in community and personal publishing models, the rise of cloud-based computing, and remote operation.

Chapter 2: .NET Platform Overview

Objectives

- Understand the high-level features of the Microsoft .NET Framework.
- Learn about specific technologies that make up the Microsoft .NET platform.
- Understand the development tools available for designing and creating .NET applications.

Overview

This chapter starts with an overview of the .NET Framework and the Common Language Runtime (CLR), followed by a series of sections that discuss the range of built-in technologies such as ASP.NET, server applications such as Microsoft® SQL Server®, development tools such as Microsoft Visual Studio®, and external libraries. Microsoft Visual Studio is the primary environment for developing .NET applications, and is available in several different versions that target specific groups involved in the full life cycle of application development. In addition to Visual Studio, Microsoft provides other development environments, such as Microsoft Expression® Studio, and external libraries that are not included in the .NET Framework.

The Microsoft .NET platform is composed of server applications, infrastructure components, run-time services used by .NET applications, and the .NET Framework, as detailed in the following table.

Table 1 Microsoft .NET platform

Category	**Technologies**
Application Infrastructure	• Common Language Runtime (CLR) • .NET Framework
Mobile	• .NET Compact Framework • ASP.NET Mobile • Silverlight Mobile
Web	• ASP.NET
Rich Internet Application (RIA)	• Microsoft Silverlight™
Rich Client	• Windows Forms • Windows Presentation Foundation (WPF)
Services	• ASP.NET Web Services (ASMX) • Windows Communication Foundation (WCF)
Collaboration / Integration / Workflow	• Windows Workflow Foundation (WF) • Microsoft Office SharePoint® Server (MOSS) • Microsoft BizTalk® Server
Web Server	• Internet Information Services (IIS)
Database Server	• Microsoft SQL Server®

.NET Framework

At a high level, the .NET Framework is composed of a virtual run-time engine, a library of classes, and run-time services used in the development and execution of .NET applications. The .NET Framework was initially released as a run-time engine and core set of classes used to build applications. Subsequent releases extended and updated the core classes, and added new technologies such as the Windows Communication Foundation (WCF).

The Base Class Library (BCL) provides a core set of classes that cover a wide range of programming requirements in a number of areas, including user interface (UI), data access, database connectivity, cryptography, numeric algorithms, and network communications.

Overlaying the BCL are core technologies for developing .NET applications. These technologies include class libraries and run-time services that are grouped by application features, such as rich client and data access. As the Microsoft .NET Platform evolves, new technologies are added on top of the core technologies, such as WCF, Windows Presentation Foundation (WPF), and Windows Workflow Foundation (WF).

Common Language Runtime (CLR)

The .NET Framework includes a virtual environment that manages the program's run-time requirements. This environment is called the Common Language Runtime (CLR) and provides the appearance of a virtual machine so that programmers do not need to consider the capabilities of the specific CPU or other hardware that will execute the program. Applications that run within the CLR are referred to as *managed applications*, and most .NET applications are developed using managed code (code that will execute within the CLR). Some applications are developed using unmanaged code such as device drivers which needs to use kernel APIs.

The CLR also provides services such as security, memory management, and exception handling.

ASP.NET

For Web applications, the ASP.NET functionality within the .NET Framework is used, hosted within Microsoft Windows Server® Internet Information Services (IIS). The .NET platform provides the following technology options for Web applications:

- **ASP.NET Web Forms**. This is the standard UI design technology for .NET Web applications. An ASP.NET Web Forms application is installed only on the Web server; no components are required on the client computer.
- **ASP.NET Web Forms with AJAX**. Asynchronous JavaScript and XML (AJAX) with ASP.NET Web Forms is used to send requests to the server asynchronously and process the responses on the client. This improves responsiveness and reduces the number of post-backs to the server. AJAX is an integral component of ASP.NET starting with the .NET Framework version 3.5.
- **ASP.NET Web Forms with Silverlight controls**. In an existing ASP.NET application, Silverlight controls can be used to improve the user experience without the requirement to write a completely new Silverlight application.

- **ASP.NET MVC**. ASP.NET MVC allows developers to use ASP.NET to easily build applications that implement the Model-View-Controller (MVC) design pattern. ASP.NET MVC supports test-driven development (TDD), and provides clear separation of concerns between UI processing and UI rendering.
- **ASP.NET Dynamic Data**. ASP.NET Dynamic Data enables the creation of ASP.NET applications that leverage Language-Integrated Query (LINQ) querying functionality. This functionality allows you to more easily model your database in your application, and query the database using LINQ queries.

Data Access

The .NET platform provides the following technology options for data access:

- **ADO.NET Core**. ADO.NET provides general features for the retrieval, update, and management of data. It includes providers for SQL Server, OLE-DB, ODBC, SQL Server Mobile, and Oracle databases.
- **ADO.NET Data Services Framework**. The ADO.NET Data Services Framework exposes data using the Entity Data Model (EDM) through RESTful Web services accessed by using HTTP. The data can be addressed directly using a Uniform Resource Identifier (URI). The Web service can be configured to return the data as plain Atom and JavaScript Object Notation (JSON) formats in ADO.NET Data Services Framework version 1, with other formats are due in subsequent releases.
- **ADO.NET Entity Framework**. This 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 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 that 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 business intelligence (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**. ADO.NET Sync Services is a provider included in the Microsoft Sync Framework synchronization for ADO.NET-enabled databases. It enables data synchronization to be built in occasionally connected applications. It periodically gathers information from the client database and synchronizes it with the server database.
- **Language-Integrated Query (LINQ)**. LINQ provides class libraries that extend C# and Microsoft Visual Basic® with native language syntax for queries. Queries can be performed against a variety of data formats, including DataSet (LINQ to DataSet), XML (LINQ to XML),

in-memory objects (LINQ to Objects), ADO.NET Data Services (LINQ to Data Services), and relational data (LINQ to Entities).Understand that LINQ is primarily a query technology supported by different assemblies throughout the .NET Framework. For example, LINQ to Entities is included with the ADO.NET Entity Framework assemblies; LINQ to XML is included with the System.Xml assemblies; and LINQ to Objects is included with the .NET core system assemblies.
- **LINQ to SQL**. 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 Framework 4.0, LINQ to SQL scenarios will be integrated and supported by the ADO.NET Entity Framework; however, LINQ to SQL will continue to be a supported technology. For more information, see the ADO.NET team blog at http://blogs.msdn.com/adonet/archive/2008/10/31/clarifying-the-message-on-l2s-futures.aspx.

Mobile Applications

The .NET platform provides the following technology options for mobile applications:
- **.NET Compact Framework**. The .NET Compact Framework is a subset of the full .NET Framework designed specifically for mobile devices. It can be used for mobile applications that must run on a device without guaranteed network connectivity.
- **ASP.NET Mobile**. ASP.NET Mobile is a subset of the ASP.NET controls plus additional functionality designed specifically for delivering content to mobile devices. ASP.NET Mobile applications can be hosted on a normal ASP.NET Web server. ASP.NET Mobile is suitable for mobile Web applications that can rely on a persistent network connection**.**
- **Silverlight Mobile**. Silverlight Mobile is a mobile device–specific version of the Silverlight client. It requires the Silverlight plug-in to be available on the mobile device. This technology can be used to port existing Silverlight applications to mobile devices, or if a richer UI is needed than possible with other options.

Rich Client

Windows-based applications are executed by the .NET Framework within Microsoft Windows®. The .NET platform provides the following technology options for rich client applications:
- **Windows Forms**. This is the standard UI design technology for the .NET Framework. Even with the availability of WPF, Windows Forms is still a good choice for applications that do not require a media-rich UI, when the developer team is familiar with it and has the relevant technical expertise.
- **Windows Forms with WPF user controls**. You can use this approach to take advantage of the more powerful UI features provided by WPF controls. You can add WPF controls to your existing Windows Forms application. However, remember that WPF controls tend to work best on higher-powered client machines.
- **Windows Presentation Foundation (WPF)**. WPF supports 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 Extensible Application Markup Language (XAML) to improve UI appearance and performance, support data binding, and define events. WPF applications can be deployed to the desktop, or displayed in a Web browser as an XAML Browser Application (XBAP).

- **XAML Browser Application (XBAP) using WPF**. The XBAP mechanism hosts a sandboxed WPF application in Microsoft Internet Explorer or Mozilla Firefox on Windows. Unlike Silverlight, you can use the full WPF framework but there are some limitations related to accessing system resources from the partial-trust sandbox. XBAP requires Microsoft Windows Vista® or both the .NET Framework version 3.0 and higher plus the XBAP browser plug-in. XBAP is a good choice for intranet-facing WPF applications due to its heavier footprint and the support for Windows only.

Rich Internet Application

For a rich Internet application (RIA), you use the Silverlight functionality of the .NET Framework, hosted within IIS. The .NET platform provides the following technology options for RIAs:

- **Silverlight**. Silverlight is a browser-optimized subset of WPF that works cross-platform and cross-browser. Compared to XBAP, Silverlight is a smaller and faster installation, but may not support all of the features of the client machine. Due to its small footprint and cross-platform support, Silverlight is a good choice for Internet-facing WPF applications.
- **Silverlight with AJAX**. Silverlight natively supports AJAX and exposes its Document Object Model (DOM) to JavaScript in the Web page. You can use this capability to support interaction between your page components and the Silverlight application.

Services

The .NET platform provides the following technologies for creating service-based applications:

- **Windows Communication Foundation (WCF)**. WCF is designed to offer a manageable approach to distributed computing and provide broad interoperability, and includes direct support for service orientation. It supports a range of protocols including Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), Microsoft Message Queuing (MSMQ), and named pipes.
- **ASP.NET Web services (ASMX)**. ASMX offers a simpler approach to distributed computing and interoperability, but supports only the HTTP protocol.

Workflow

The .NET platform provides the following technology options for implementing workflows:

- **Windows Workflow Foundation (WF)**. WF is a foundational technology that allows you to implement workflow. WF is a toolkit for professional developers and independent software vendors (ISVs) who want to build a sequential or state machine-based workflow. WF supports the following types of workflow: Sequential, State-Machine, Data-Driven, and Custom. You can create workflows using the Windows Workflow (WF) Designer in Visual Studio. WF is integrated with WCF in .NET version 3.5 to provide WCF-based services for workflow.

- **Microsoft Office SharePoint Server (MOSS)**. MOSS is a server application that provides a platform for content management and collaboration. MOSS workflow is based on a version of Windows Workflow Foundation (WF). It is a solution for human workflow and collaboration in the context of a SharePoint server. You can create basic workflows using administrative interfaces that are included in the SharePoint Central administration tool. To extend or customize the SharePoint workflow solutions, you can use Visual Studio with WF. You can also customize workflow markup using either the Microsoft Office SharePoint Designer or the Windows Workflow Foundation (WF) Designer in Visual Studio**.**
- **BizTalk Server.** BizTalk uses a workflow engine geared toward orchestration, such as enterprise integration with system-level workflows. You can define the overall design and flow of loosely coupled, long-running business processes within applications, and between applications, using BizTalk Orchestration services.

Note: MOSS and BizTalk server are not part of the .NET Framework or Visual Studio; these are independent products, but part of the overall .NET platform.

Web Server – Internet Information Services (IIS)

The Microsoft platform includes Internet Information Services (IIS), which provides full-scale support for Internet publishing, including transport services, client applications, administrative tools, database and application connectivity, and encrypted communication. IIS supports the following services:

- **World Wide Web Service**. This service provides all the features required for hypertext document publishing, and delivering other types of content that use the HTTP protocol. It provides high performance, compression, extensive configurability, and supports a range of security and authentication options.
- **File Transfer Protocol (FTP) Service**. This service allows you to receive and deliver files using the FTP protocol. However, authentication is limited to the Basic method.
- **Gopher Service**. This service supports a distributed document search and retrieval network protocol. It is rarely used today.
- **Internet Database Connector**. This is an integrated gateway and template scripting mechanism for the World Wide Web service to access Open Database Connectivity (ODBC) databases. Generally superseded by new data-access and scripting technologies such as ASP.NET and ASP.NET Data Services.
- **Secure Sockets Layer (SSL) Client/Server**. This provides a mechanism to support encrypted communication over the HTTP protocol, allowing clients and servers to communicate more securely than when sending content as plain text.
- **Internet Service Manager Server**. This is an administration console and associated tools that provide local and remote administration features for IIS.
- **Integration with ASP.NET**. IIS 7.0 and later is specifically designed to integrate closely with ASP.NET to maximize performance and minimize server load when using ASP.NET to create and deliver content.

Database Server – SQL Server

A relational database is a common approach for storing and accessing data in an enterprise application. The .NET platform provides SQL Server as the database engine for your applications. SQL Server is available in several variants, from a single-instance, local database (SQL Server Express) scaling to enterprise-level applications through SQL Server Enterprise Edition.

The data access technologies that are part of the .NET Framework allow you to access data in any version of SQL Server, so you do not need to modify your application if you want to scale up to a more powerful version.

Visual Studio Development Environment

The .NET platform provides a comprehensive development environment known as the Visual Studio Team System. You can use the language of your choice within Visual Studio Team System to write applications that target the .NET Framework. As an integrated development environment (IDE), it provides all the tools you require to design, develop, debug, and deploy Windows, Web, Mobile, and Office-based solutions. Visual Studio is organized into several different versions, with some versions targeted at specific groups such as architects, testers, and others involved in the full life cycle of application development. You can install multiple versions side-by-side to obtain the required combination of features. In addition to Visual Studio, Microsoft provides other development environments such as Expression Studio, and external libraries that are not included in the .NET Framework.

Other Tools and Libraries

In addition to Visual Studio, other tools and frameworks are available to speed development or facilitate specific types of application development. Examples are:

- System Center, which provides a set of tools and environments for enterprise-level application monitoring, deployment, configuration, and management. For more information, see *Microsoft System Center* at http://www.microsoft.com/systemcenter/en/us/default.aspx.
- Expression Studio, which provides tools aimed at graphical designers for creating rich interfaces and animations. For more information, see *Microsoft Expression* at http://www.microsoft.com/expression/products/Overview.aspx?key=studio.

patterns & practices solution Assets

- Enterprise Library contains a series of application blocks that address cross-cutting concerns. For more information, see *Enterprise Library* at http://msdn.microsoft.com/en-us/library/cc467894.aspx.
- Software Factories speed development of specific types of application such as Smart Clients, WPF applications, and Web Services. For more information, see *patterns & practices: by Application Type* at http://msdn.microsoft.com/en-gb/practices/bb969054.aspx.

Additional Resources

- For more information about the .NET Framework, see *.NET Framework 3.5 Overview* at http://msdn.microsoft.com/en-us/library/a4t23ktk.aspx.
- For more information about the .NET Framework, see *Overview of the .NET Framework* at http://msdn.microsoft.com/en-us/library/zw4w595w(VS.71).aspx.
- For more information about the .NET Framework, see *Overview of the .NET Compact Framework* at http://msdn.microsoft.com/en-us/library/w6ah6cw1(VS.80).aspx.
- For more information about Web Services, see *Windows Communication Foundation* at http://msdn.microsoft.com/en-us/library/ms735119.aspx.
- For more information about Web Services, see *XML Web Services Using ASP.NET* at http://msdn.microsoft.com/en-us/library/ba0z6a33.aspx.
- For more information about workflow services, see *Workflows in Office SharePoint Server 2007* at http://msdn.microsoft.com/en-us/library/ms549489.aspx.
- For more information about workflow services, see *Windows Workflow Foundation (WF)* at http://msdn.microsoft.com/en-us/netframework/aa663328.aspx,
- For more information on data access, see *Data Platform Development* at http://msdn.microsoft.com/en-gb/data/default.aspx.
- For more information about the IIS Web server, see *A High-Level Look at Microsoft Internet Information Server* at http://msdn.microsoft.com/en-us/library/ms993571.aspx.
- For more information about SQL Server, see *SQL Server* at http://msdn.microsoft.com/en-gb/sqlserver/default.aspx.
- For more information about Visual Studio Team System, see *Visual Studio 2008 Overview* at http://msdn.microsoft.com/en-us/vstudio/products/bb931331.aspx.

Chapter 3: Architecture and Design Guidelines

Objectives

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

Overview

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

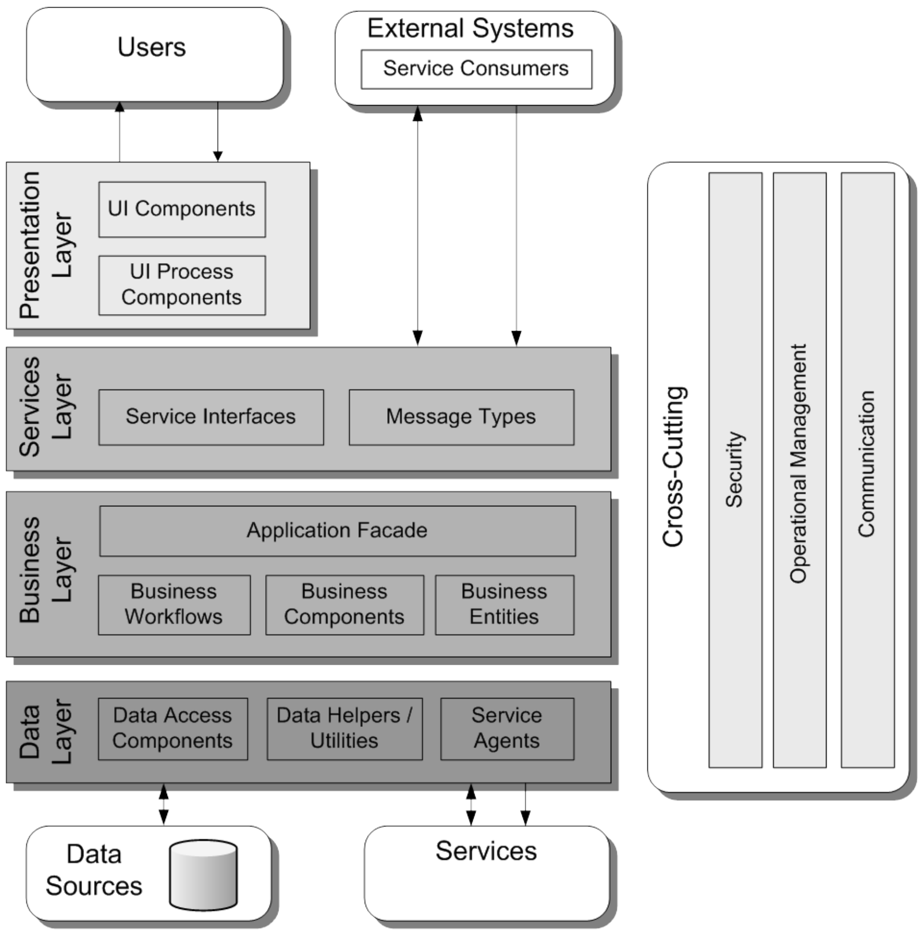

Figure 1 Common application architecture

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

Key Design Principles

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

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

Design Considerations

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

Follow these guidelines when designing an application:

- **Avoid doing all your design upfront**. If you are not clear with requirements or if there is the possibility of design evolution, it might be a good idea not to do complete design upfront. Instead, evolve the design as you progress through the project.
- **Separate the areas of concern**. Break your application into distinct features that overlap in functionality as little as possible. The main benefit of this approach is that a feature or functionality can be optimized independently of other features or functionality. Also, if one feature fails, it will not cause other features to fail as well, and they can run independently

of one another. This approach also helps to make the application easier to understand and design, and facilitates management of complex interdependent systems.

- **Each component or module should have a single responsibility**. Each component or module should be responsible for only one specific feature or functionality. This makes your components cohesive and makes it easier to optimize the components if a specific feature or functionality changes.
- **A component or an object should not rely on internal details of other components or objects**. Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if needed, route it to appropriate subcomponents or other components. This helps in developing an application that is more maintainable and adaptable.
- **Do not duplicate functionality within an application**. There should be only one component providing a specific functionality—this functionality should not be duplicated in any other component. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.
- **Identify the kinds of components you will need in your application**. The best way to do this is to identify patterns that match your scenario and then examine the types of components that are used by the pattern or patterns that match your scenario. For example, a smaller application may not need business workflow or UI processing components.
- **Group different types of components into logical layers**. Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers.
- **Keep design patterns consistent within each layer**. Within a logical layer, the design of components should be consistent for a particular operation. For example, if you choose to use the Table Data Gateway pattern to create an object that acts as a gateway to tables or views in a database, you should not include another pattern such as Repository, which uses a different paradigm for accessing data and initializing business entities.
- **Do not mix different types of components in the same logical layer**. For example, the UI layer should not contain business-processing components, but instead should contain components used to handle user input and process user requests.
- **Determine the type of layering you want to enforce**. In a strict layering system, components in layer A cannot call components in layer C; they always call components in layer B. In a more relaxed layering system, components in a layer can call components in other layers that are not immediately below it. In all cases, you should avoid upstream calls and dependencies.
- **Use abstraction to implement loose coupling between layers**. This can be accomplished by defining interface components such as a façade with well-known inputs and outputs that translate requests into a format understood by components within the layer. In addition, you can also use **Interface** types or abstract base classes to define a common interface or shared abstraction (dependency inversion) that must be implemented by interface components.
- **Do not overload the functionality of a component**. For example, a UI processing component should not contain data access code. A common anti-pattern named Blob is

often found with base classes that attempt to provide too much functionality. A Blob object will often have hundreds of functions and properties providing business functionality mixed with cross-cutting functionality such as logging and exception handling. The large size is caused by trying to handle different variations of child functionality requirements, which requires complex initialization. The end result is a design that is very error-prone and difficult to maintain.

- **Understand how components will communicate with each other**. This requires an understanding of the deployment scenarios your application will need to support. You need to determine if communication across physical boundaries or process boundaries should be supported, or if all components will run within the same process.
- **Prefer composition over inheritance**. Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes. This also reduces the inheritance hierarchies, which can become very difficult to deal with.
- **Keep the data format consistent within a layer or component**. Mixing data formats will make the application more difficult to implement, extend, and maintain. Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation.
- **Keep cross-cutting code abstracted from the application business logic as much as possible**. Cross-cutting code refers to code related to security, communications, or operational management such as logging and instrumentation. Attempting to mix this code with business logic can lead to a design that is difficult to extend and maintain. Changes to the cross-cutting code would require touching all of the business logic code that is mixed with the cross-cutting code. Consider using frameworks that can help to implement the cross-cutting concerns.
- **Be consistent in the naming conventions used**. Check to see if naming standards have been established by the organization. If not, you should establish common standards that will be used for naming. This provides a consistent model that makes it easier for team members to evaluate code they did not write, which leads to better maintainability.
- **Establish the standards that should be used for exception handling**. For example, you should always catch exceptions at layer boundaries, you should not catch exceptions within a layer unless you can handle them in that layer, and you should not use exceptions to implement business logic. The standards should also include policies for error notification, logging, and instrumentation when there is an exception.

Architecture Frame

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

Table 1 Architecture Frame

Area	Key issues
Authentication 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 coupled by dependencies making development, testing, and maintenance more difficult • Dependency changes between modules, forcing code recompilation and module redeployment • Difficulties in dynamic UI layout and update due to hard-coded dependencies • Difficulty in dynamic module loading due to hard-coded 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	• Failing to an unstable state • Revealing sensitive information to the end user • Using exceptions to control application flow • Not logging sufficient details about the exception
Layering	• Incorrect grouping of components within a layer • Not following layering and dependency rules • Not considering the physical distribution of layers

Area	Key issues
Logging and Instrumentation	• Lack of logging and instrumentation • Logging and instrumentation that is too fine-grained • Not making logging and instrumentation an option that is configurable at run time • Not suppressing and handling logging failures • Not logging business-critical functionality
State Management	• Using an incorrect state store • Not considering serialization requirements • Not persisting state when required
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 unrelated functionality
Validation	• Lack of validation across trust boundaries • Failure to validate for range, type, format, and length • Not reusing validation logic
Workflow	• Not considering management requirements • Choosing an incorrect workflow pattern • Not considering exception states and how to handle them

Authentication

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

Consider the following guidelines when designing an authentication strategy:

- Identify your trust boundaries and authenticate users and calls across the trust boundaries. Consider that calls might need to be authenticated from the client as well as from the server (mutual authentication).
- If you have multiple systems within the application that 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. Failure to design and implement a good authorization strategy can make your application vulnerable to information disclosure, data tampering, and elevation of privileges.

Consider the following guidelines when designing an authorization strategy:

- Identify your trust boundaries and authorize users and callers across the 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.

Consider following guidelines when designing a caching strategy:

- Do not cache volatile data.
- Consider using ready-to-use cache data when working with an in-memory cache. For example, use a specific object instead of caching raw database data.
- Do not cache sensitive data unless you encrypt it.
- If your application is deployed in Web farm, avoid using local caches that need to be synchronized; instead consider using a transactional resource manager such as Microsoft® 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.

Consider the following guidelines when designing communication mechanisms:

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

- Consider using Microsoft Message Queuing (MSMQ) to queue messages for later delivery in case of system or network interruption or failure. MSMQ can perform transacted message delivery and supports reliable once-only delivery.

Composition

Composition is the process used to define how interface components in a UI are structured to provide a consistent look and feel for the application. One of the goals of UI 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.

Consider the following guidelines when designing for composition:

- Avoid using dynamic layouts because 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 database concurrency, you can choose between an optimistic model, where the last update applied is valid, or a pessimistic model, where updates can only be applied to the latest version. Transactions can be executed within the database, or they can be executed in the business layer of an application. Where you choose to implement transactions depends on your transactional requirements. Concurrency should also be considered when accessing static data within the application or when using threads to perform asynchronous operations. Static data is not thread-safe, which means that changes made in one thread will affect other threads using the same data.

Consider the following guidelines when designing for concurrency and transactions:

- If you have business-critical operations, consider wrapping them in transactions.
- Use connection-based transactions when accessing a single data source.
- 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.

- Updates to shared data should be mutually exclusive, which is accomplished by applying locks or by using thread synchronization. This will prevent two threads from attempting to update shared data at the same time.
- Use synchronization support provided by collections when working with static or shared collections.

Configuration Management

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

Consider the following guidelines when designing for configuration management:
- Consider using least-privileged process and service accounts.
- Categorize the configuration items into logical sections if your application has multiple tiers.
- If your server application runs in a Web farm, decide which parts of the configuration are shares and which parts are specific to the machine on which the application is running. 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 logging only represents high cohesion.

Consider the following guidelines when designing for coupling and cohesion:
- Partition application functionality into logical layers.
- Design for loose coupling between layers. Consider using abstraction to implement loose coupling between layers with interface components, common interface definitions, or shared abstraction. *Shared abstraction* is where concrete components depend on abstractions and not on other concrete components (the principle of dependency inversion).
- Design for high cohesion. Components should contain only functionality that is specifically related to that component.
- Know the benefits and overhead of loosely coupled interfaces. While loose coupling requires more code, the benefits include a shortened dependency chain and a simplified build process.

Data Access

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

Consider the following guidelines when designing data access components:

- Avoid coupling your application model directly to the database schema. Instead, you should consider using an abstraction or mapping layer between the application model and database schema.
- 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. Failure to do so can make your application vulnerable to Denial of Service (DoS) attacks, and may also reveal sensitive and critical information. Raising and handling exceptions is an expensive process. It is important that the design also takes into account the performance considerations. A good approach is to design a centralized exception-management and logging mechanism, and to consider providing access points within your exception-management system to support instrumentation and centralized monitoring that assists system administrators.

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

- Do not catch 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.

Consider the following guidelines when designing layers:

- Layers should represent a logical grouping of components. For example, use separate layers for UI, business logic, and data access components.
- Components within a layer should be cohesive. In other words, the business layer components should provide only operations related to application business logic.
- 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. Failure to do so can make your application vulnerable to repudiation threats, where users deny their actions. Log files may be required for legal proceedings to prove the wrongdoing of individuals. You should audit and log activity across the layers of your application. Using logs, you can detect suspicious activity, which can provide early indication of a serious attack. Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access, and by the same routines that access the resource. Instrumentation can be implemented by using performance counters and events to give administrators information about the state, performance, and health of an application.

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

State Management

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

Consider the following guidelines when designing a state management mechanism:

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

Consider the following guidelines when designing the application structure:

- 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 subnets.
- 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.

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

- Design for a consistent navigation experience. Use composite patterns for the look and feel, and controller patterns such as Model-View-Controller (MVC), 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 UI guidelines. In many cases, an organization will have published guidelines to which you should adhere.

Validation

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

Consider the following guidelines when designing a validation mechanism:

- Identify your trust boundaries, and validate all inputs across the trust boundaries.
- Centralize your validation approach, if it can be reused.
- Constrain, reject, and sanitize user input. In other words, assume that all user input is malicious.
- Validate input data for length, format, type, and range.
- Do not rely only on client-side validation for security checks. Instead, use client-side validation to give the user feedback and improve the user experience. Because client-side validation can be bypassed when attacking the server, use server-side validation to check for malicious input.

Workflow

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

Consider the following guidelines when designing a workflow component:

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

Pattern Map

Table 2 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, reduced 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 that 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 on data tiers, 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.

PART II

Design

In This Part:

- ▶ **Designing Your Architecture**
- ▶ **Deployment Patterns**
- ▶ **Architecture Styles**
- ▶ **Quality Attributes**
- ▶ **Communication Guidelines**

Chapter 4: Designing Your Architecture

Objectives

- Understand the inputs required for the architecture design process.
- Understand the typical results of the architecture design process.
- Learn the key steps to iteratively design a suitable architecture for your application.
- Learn the guidelines to follow when designing the architecture.
- Learn an effective approach for reviewing the architecture.

Overview

A successful approach to architecture design requires you to follow a series of steps that ensure that you consider all of the relevant factors affecting the architecture, and then repeat these steps as you refine your design. This chapter recommends a series of five main steps, each of which breaks down into individual processes that will be explained in greater detail throughout the remainder of the guide. These steps are meant to be iterative; you will produce candidate solutions that you can further refine by repeating the steps, finally creating the architecture design that best fits your application.

Input

The following list represents assets that are useful to have in hand when designing your architecture:

- Use cases and usage scenarios
- Functional requirements
- Non-functional requirements including quality attributes such as performance, security, reliability, and others.
- Technological requirements
- Target deployment environment
- Constraints

Output

The steps in this chapter should result in the following assets:

- Architecturally significant use cases
- Architecture hotspots
- Candidate architectures

Steps

Figure 1 shows the five major steps for designing your architecture.

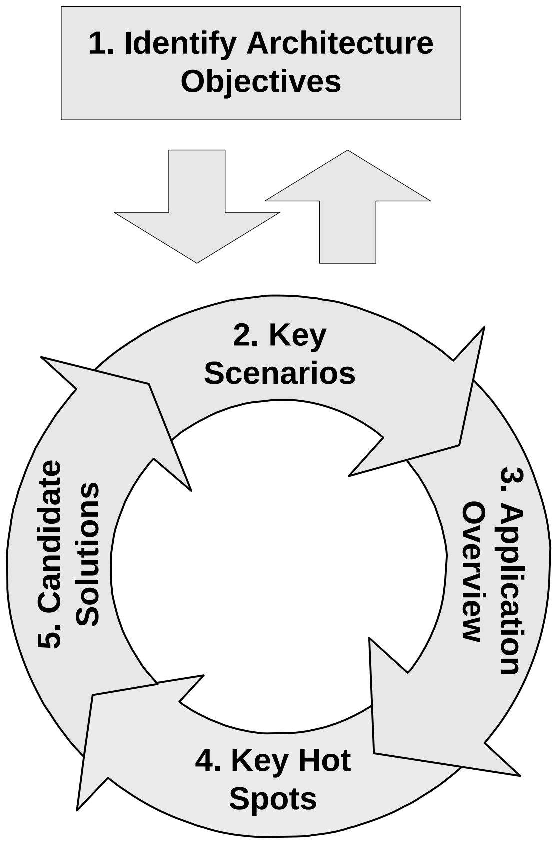

Figure 1 Core Architecture Design Activities

These steps are:

- **Step 1: Identify Architecture Objectives**. Clear objectives help you to focus on your architecture and on solving the right problems in your design. Precise objectives help you to determine when you have completed the current phase, and when you are ready to move to the next phase.
- **Step 2**: **Key Scenarios**. Use key scenarios to focus your design on what matters most, and to evaluate your candidate architectures when they are ready.
- **Step 3: Application Overview**. Understand your application type, deployment architecture, architecture styles, and technologies in order to connect your design to the real world in which the application will have to operate.
- **Step 4: Key Hotspots**. Identify key hotspots based on quality attributes and the architecture frame. These are the areas where mistakes are most often made when designing an application.
- **Step 5: Candidate Solutions**. Create a candidate architecture or architectural spike and evaluate it against your key scenarios, hotspots, and deployment constraints.

This architectural process is meant to be an iterative and incremental approach. Your first candidate architecture will be a high-level design that you can test against key scenarios, requirements, known constraints, quality attributes, and the architecture frame. As you refine your candidate architecture, you will learn more details about the design and will be able to further flesh out key scenarios, your application overview, and your approach to hotspots.

You can iteratively flesh out your architecture as you work through your design and discover more details that impact your architecture. You should not try to build your architecture in a single iteration. Each iteration should add more detail. Don't get lost in the details, but instead focus on the big steps and build a framework on which you can base your architecture and design.

Step 1: Identify Architecture Objectives

Overview

Architecture objectives are the goals and constraints that shape your architecture and design process, scope the exercise, and help you determine when you are finished. Consider the following key points as you identify your architecture objectives:

- **Understand your architecture goals at the start**. The amount of time you spend in each phase of architecture and design will depend on these goals. For example, are you building a prototype, testing potential paths, or embarking on a long-running architectural process for a new application?
- **Understand who will consume your architecture**. Determine if your architecture will be used by other architects, by developers and testers, by operations staff, or by management. Consider the needs of your audience to make your architecture more successful and impactful.
- **Understand your constraints**. Understand your technology options and constraints, usage constraints, and deployment constraints. Understand your constraints at the start so that you do not waste time or encounter surprises later in your application development process.

Scope and Time

Based on your high-level goals for the architecture process, you can scope the amount of time to spend on your design. For instance, a prototype might only require a few days to design, while a fully detailed architecture for a complex application could potentially take months to complete. Use your understanding of the objectives to determine how much time and energy to spend on each step and to gain an understanding of what the end result will look like.

Use your architecture objectives to clearly define the purpose and priorities of your architecture. Possible purposes might include:

- A complete application design
- Building of prototypes
- Identification of key technical risks

- Testing of potential paths
- Shared models and understanding of system

Each of these purposes will result in a different emphasis during design and varying time commitment, and will impact the results and design documentation that emerge from the process. For instance, if you want to identify key risks in your authentication architecture, you will spend much of your time and energy identifying authentication scenarios, constraints on your authentication architecture, and possible authentication technology choices. However, if you are building a larger design, authentication will be only one of many other concerns you address and document solutions for.

Some examples of architecture objectives are:
- Build a prototype to get feedback on the order-processing UI for a Web application.
- Test three possible ways to map location data to search results.
- Build a customer order-tracking application.
- Design the authentication and authorization architecture for an application for the purposes of security review.

Step 2: Key Scenarios

Understand your key scenarios in order to shape your application to meet these scenarios, and later to use as tests of candidate architectures and architectural spikes. Key scenarios are those that are considered the most important scenarios for the success of your application. Key scenarios can be defined as any scenario that meets one of the following criteria:
- It is an architecturally significant use case.
- It represents the intersection of quality attributes with functionality.
- It represents a tradeoff between quality attributes.

For example, your authentication strategy is a key scenario because it is an intersection of a quality attribute (security) with functionality (how a user logs into your system). Another key scenario is how your security requirements impact your application's performance, because it represents the intersection of two quality attributes.

Architecturally Significant Use Cases

Architecturally significant use cases have impact across many aspects of your design. These use cases are especially important in shaping the success of your application. They are:
1. **Business-Critical**. The use case is business-critical, has a high usage level compared to other features, or implies high technical or technological risk.
2. **High Impact**. The use case intersects with both functionality and quality attributes, or represents a cross-cutting concern that has an end-to-end impact across the layer and tiers of your application. An example might be a Create, Read, Update, Delete (CRUD) operation that is security-sensitive.

Architecturally significant use cases are those that are important for the success and acceptance of the deployed application, and that exercise enough of the design to be useful in evaluating the architecture. These use cases can be categorized as system scenarios and user scenarios. *System scenarios* are those that primarily impact the internal operations of the application and infrastructure, such as message communication between layers, connecting to data stores, and performing input and data validation. *User scenarios* are those initiated by or controlled by the user, such as creating an order, viewing a product, or updating a customer record.

User Stories, System Stories, and Business Stories

Use stories from multiple perspectives to help expose the key scenarios for your system. *User stories* describe how your users will interact with the system. *System stories* describe how the system will work and organize its functionality. *Business stories* describe how the system will meet business needs or work within business constraints. Examples of these stories include:

- **User story**. A user places an order and gets a license key within 30 seconds.
- **System story**. The license process system will be able to process 100 license requests per minute.
- **Business story**. The system will require no more than two servers for deployment.

Use system stories and user stories to measure your requirements against specific, testable instances of use cases. A good story will intersect the user view, the system view, and the business view of the architecture. Use stories to test your design and determine where any breaking points may be. For instance, you would create stories around usage, and evaluate against your quality attributes. You should be able to complete development of the features to implement a story within a single iteration. You might need to develop new stories as you create and update your architecture model.

Consider the following when planning your stories:

- Early in the project, reduce risk by creating a candidate architecture that supports architecturally significant end-to-end scenarios that exercise all layers of the architecture.
- Using your architecture model as a guide, make changes to your architecture, design, and code to meet your scenarios, functional requirements, technological requirements, quality attributes, and constraints.
- Create an architecture model based on what you know at the time, and define a list of questions that must be addressed in subsequent stories and iterations.
- After you make sufficient significant changes to the architecture and design, consider creating a story that reflects these changes. Batch together the small changes in the architecture and design and address them together.

Step 3: Application Overview

Summary of Steps

Build an overview of what your application will look like when it is complete. The application overview serves to make your architecture more real, connecting it to real-world constraints and decisions. An application overview consists of the following steps:

1. **Determine your application type**. First, determine what type of application you are building. Is it a mobile application, a rich client, a rich Internet application, a service, a Web application, or some combination of these types?
2. **Understand your deployment constraints**. Next, understand your targeted deployment environment and determine what impact it will have on your architecture.
3. **Identify important architecture styles**. Determine which architecture styles you will be using in your design. Will you build service oriented architecture (SOA), client/server, layered, message-bus, or a combination of styles?
4. **Determine relevant technologies**. Finally, identify the relevant technology choices based on your application type and other constraints, and determine which technologies you will leverage in your architecture.

Application Type

Choosing the right application type is the key part of the process of designing and architecting an application. Your choice of an appropriate application type is governed by your specific requirements and infrastructure limitations. The following considerations will help you to choose the appropriate application type.

Key Application Types

- **Mobile applications**. These applications can be developed as thin client or rich client applications. Rich client mobile applications can support disconnected or occasionally-connected scenarios, while Web or thin client applications support connected scenarios only. The device resources may prove to be a constraint when designing mobile applications.
- **Rich client applications**. These applications are usually developed as stand-alone applications with a graphical user interface (GUI) that displays data using a range of controls. Rich client applications can be designed to support disconnected and occasionally-connected scenarios because the application runs on the client machine.
- **Rich Internet applications**. These applications can be developed to support multiple platforms and multiple browsers, displaying rich media or graphical content. Rich Internet applications run in a browser sandbox that restricts access to devices on the client.
- **Services applications**. These applications aim to achieve loose coupling between the client and the server. Services expose complex functionality and allow clients to access the service from a local or remote machine. Service operations are called using Extensible Markup Language (XML)–based message schemas passed over a transport mechanism.
- **Web applications**. These applications typically support connected scenarios, and are developed to support multiple browsers and multiple operating system platforms.

Choosing Application Types

Choose the appropriate application type by considering your requirements and the infrastructure limitations. Use the table below to make an informed choice based on the benefits and considerations for each application type.

Table 1 Application type considerations

Application type	Benefits	Considerations
Mobile Applications	• Can support handheld devices. • Provide availability and ease of use for out-of-office users • Can support offline and occasionally-connected scenarios.	• Input and navigation limitations • Limited screen display area
Rich Client Applications	• Can leverage client resources. • Provide better responsiveness, rich UI functionality, and improved user experience. • Provide highly dynamic and responsive interaction. • Can support offline and occasionally connected applications.	• Deployment complexity; however, a range of installation options are available, such as ClickOnce, Windows Installer, and XCOPY • Can be challenging to version over time • Platform-specific
Rich Internet Applications (RIA)	• Provide the same rich UI capability as rich clients. • Provide support for rich media and graphics display. • Simple deployment and the distribution capabilities (reach) of Web clients.	• Larger application footprint on the client machine compared to a Web application • Restrictions on leveraging client resources compared to a rich client application • Requires deployment of the .NET or Microsoft Silverlight® run time on the client
Services Applications	• Provide loosely coupled interactions between client and server. • Can be consumed by different and unrelated applications. • Supports interoperability.	• No UI support • Client is dependent on network connectivity
Web Applications	• Has broad reach, and a standards-based UI across multiple platforms. • Offers ease of deployment and change management.	• Dependency on network connectivity (must be connected all of the time) • Difficulty in providing a rich UI

Deployment Scenario

When you design your application architecture, you must take into account corporate policies and procedures, together with the infrastructure on which you plan to deploy your application. If the target environment is fixed or inflexible, your application design must reflect restrictions that exist in that environment. Your application design must also take into account Quality-of-

Service (QoS) attributes such as security and maintainability. Sometimes you must make design tradeoffs due to protocol restrictions and network topologies.

Identify the requirements and constraints that exist between the application architecture and infrastructure architecture early in the design process. This helps you to choose an appropriate deployment topology and to resolve conflicts between the application and infrastructure architecture early in the process.

Distributed and Non-distributed Architectures

Applications are typically deployed in one of two ways: *non-distributed deployment*, where all of the functionality and layers reside on a single server except for data storage functionality; or *distributed deployment*, where the layers of the application reside on separate physical tiers. In most cases, the recommendation is to use non-distributed deployment. Whenever a process must cross physical boundaries, performance is affected because the data must be serialized. However, there are some cases where you need to split functionality across servers. In addition, depending on where servers are located, you can often choose communication protocols that are optimized for performance.

Non-distributed Deployment

With the non-distributed architecture, presentation, business, and data access code are logically separated but are physically located in a single server process. Figure 2 illustrates the non-distributed scenario.

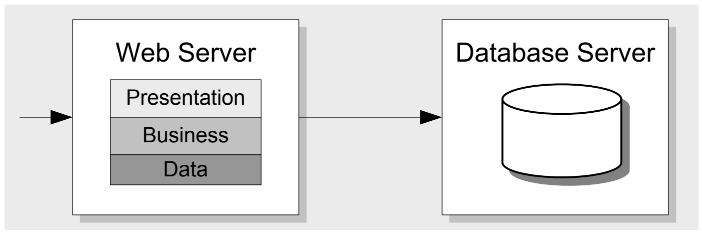

Figure 2 Non-distributed deployment

Advantages

- Non-distributed architecture is less complex than distributed architecture.
- Non-distributed architecture has performance advantages gained through local calls.

Disadvantages

- With non-distributed architecture, it is difficult to share business logic with other applications.
- With non-distributed architecture, server resources are shared across layers. This can be good or bad. Layers may work well together and result in optimized usage because one of them is always busy. However, if one layer requires disproportionately more resources, another layer may be starved of resources.

Distributed Deployment

Distributed deployment allows you to separate the layers of an application on different physical tiers. Figure 3 illustrates the distributed scenario.

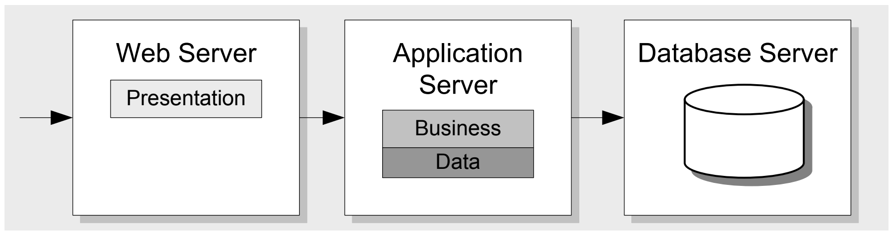

Figure 3 Distributed deployment

Advantages

- Distributed architecture has the ability to scale out and load-balance business logic independently.
- Distributed architecture has separate server resources that are available for separate layers.
- Distributed architecture is flexible.

Disadvantages

- Distributed architecture has additional serialization and network latency overheads due to remote calls.
- Distributed architecture is potentially more complex and more expensive in terms of total cost of ownership (TCO).

Architecture Styles

An architecture style is a set of principles. You can think of it as a coarse-grained pattern that provides an abstract framework for a family of systems. Each style defines a set of rules that specify the kinds of components you can use to assemble a system, the kinds of relationships used in their assembly, constraints on the way they are assembled, and assumptions about the meaning of how you put them together. An architecture style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems.

There are many factors that influence the architecture styles that you follow. These include the capacity of your organization for design and implementation, the capabilities and experience of developers, and the infrastructure constraints and available deployment scenarios.

You will typically combine multiple styles to define a complete architecture. For example, a layered architecture can be used with component-based, object-oriented, or service-oriented architecture styles. The following are some points to consider when choosing architecture styles.

Architecture Style Frame

Architecture styles can be organized by their key focus area. The following table lists the major areas of focus and the corresponding architecture styles.

Table 2 Architecture style frame

Category	Architecture styles
Deployment	Client/server, 3-tier, *N*-tier
Structure	Component-Based, Object-Oriented, Layered Architecture, Model-View-Controller (MVC)
Domain	Domain Model, Gateway
Communication	Service-Oriented Architecture (SOA), Message Bus, Pipes and Filters

These styles are not exclusive, and you will often choose multiple overlapping styles to suit your architectural needs. For example, you might design using object-oriented code principles, organize as a layered architecture, use the Domain model for data access, communicate with services using SOA, and deploy using an N-tier style.

Key Architecture Styles

The following table lists common architecture styles, along with a brief description of each style or pattern. Later sections provide more detailed descriptions of each style, as well as guidance to help you choose the appropriate style for your application.

Table 3 Architecture styles

Architecture style	Description
Client-Server	Segregates the system into two computer programs where one program, the client, makes a service request to another program, the server.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that are location-transparent and expose well-defined communication interfaces.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message-Bus	Is a software system that can receive and send messages that are based on a set of known formats, so that systems can communicate with each other without needing to know the actual recipient.
Model-View-Controller (MVC)	Separates the logic for managing user interaction from the UI view and from the data with which the user works.
N-tier / 3-tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	A programming style based on division of tasks for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to that object.
Service-Oriented Architecture (SOA)	Represents applications that expose and consume functionality as a service using contracts and messages.

You will typically combine multiple styles to define a complete architecture. For example, a layered architecture can be used with component-based, object-oriented, or SOA styles. Consider the following points when choosing architecture styles.

Client/Server
Consider the client/server architecture style if:
- Your application is server-based and will support many clients.
- You are creating Web-based applications exposed through a Web browser.
- You are implementing business processes that will be used by people throughout the organization.
- You want to centralize data storage, backup, and management functions.
- Your application must support different client types and different devices.

Component-based
Consider the component-based architecture style if:
- You already have suitable components, or can obtain suitable components from third-party suppliers.
- Your application will predominantly execute procedural style functions, perhaps with little or no data input.
- Your application is relatively simple, and does not warrant a full layered architecture.
- Your application has specific requirements that do not include a UI or business processes.
- You want to be able to combine components written in different code languages.
- You want to create a pluggable architecture that allows you to easily replace and update individual components.

Layered
Consider the layered architecture style if:
- Your application is complex, and you want to mitigate that complexity by grouping functionality into different areas of concern.
- You want to improve the maintainability and extensibility of the application, by minimizing dependencies.
- You already have applications that expose suitable business processes through service interfaces.
- Your application must support different client types and different devices.
- You want to implement complex and/or configurable business rules and processes.

Message-bus
Consider the message-bus architecture style if:
- You have existing applications that interoperate with each other to perform tasks.
- You are implementing a task that requires interaction with external applications.
- You are implementing a task that requires interaction with applications hosted in a different environment.

- You have existing applications that perform specific tasks, and you want to combine those tasks into a single operation.
- You want to perform operations asynchronously, such as order processing or workflow.
- You are implementing a publisher/subscriber application.

Model-View-Controller (MVC) Architectural Style

Consider the Model-View-Controller architectural style if:

- You want improved testability and simpler maintenance of UI functionality.
- You want to separate the task of creating the UI from the logic code that drives it.
- Your UI view does not contain any request-processing code.
- Your UI processing code does not implement any business logic.

3-Tier/N-Tier Architectural Style

Consider either the N-tier or the 3-tier architectural style if:

- The processing requirements of the layers in the application differ. Processing in one layer could absorb sufficient resources to slow the processing in other layers.
- The security requirements of the layers in the application may differ. For example, the presentation layer will not store sensitive data, while this may be stored in the business and data layers.
- You want to be able to share business logic between applications.
- You have sufficient hardware to allocate the required number of servers to each tier.

Consider the 3-tier architectural style if:

- You are developing an intranet application where all servers are located within the private network.
- You are developing an Internet application, and security requirements do not restrict implementing business logic within the public-facing Web or application server.

Consider using more than three tiers if:

- Security requirements dictate that business logic cannot be deployed to the perimeter network.
- The application makes heavy use of resources and you want to offload that functionality to another server

Object-oriented

Consider the object-oriented architecture style if:

- You want to model the application based on real-world objects and actions.
- You already have suitable objects and classes that match the design and operational requirements.
- You need to encapsulate logic and data together in reusable components.

SOA

Consider the service-oriented architecture style if:

- You have access to suitable services or can purchase suitable services exposed by a hosting company.
- You want to build applications that compose a variety of services into a single UI.
- You are creating Software plus Service (S+S), Software as a Service (SaaS), or cloud-based applications.
- You need to support message-based communication between segments of the application.
- You need to expose functionality in a platform-independent way.
- You want to take advantage of federated services, such as authentication.
- You want to expose services that are discoverable through directories and can be used by clients that have no prior knowledge of the interfaces.

Technologies

When choosing technologies for your application, the key factors to consider are the type of application you are developing, and your preferred options for application deployment topology and architectural styles. The choice of technologies will also be governed by organization policies, infrastructure limitations, resource skills, and so on. For example, if you are building an SOA-style application, then Windows Communication Foundation (WCF) is a good choice. If you are building a Web application that will make calls into a WCF service, then ASP.NET is a good choice. Your technology choice is directly tied to your application type.

Consider the following questions:

- Which technologies help you support your chosen architectural styles?
- Which technologies help you support your application type?
- Which technologies help you support key quality attributes for your application?

Mobile

The following presentation-layer technologies are available for creating mobile applications:

- **.NET Compact Framework**. You can use the .NET Compact Framework to create a rich client mobile application that supports connected or occasionally connected scenarios.
- **ASP.NET mobile controls**. You can use ASP.NET mobile controls to create a thin client mobile application. ASP.NET mobile controls is a set of server-side controls and special page classes that render output specific to the type of device accessing the application.
- **Silverlight**. You can use Silverlight for Mobile Devices to provide rich media support and an improved user experience.

Rich Client

The following presentation-layer technologies are available for creating rich client applications:

- **Windows Forms**. You can use Windows Forms to create applications that provide rich functionality and user experience by utilizing the resources of the client PC.
- **Windows Forms with WPF user controls**. You can use Windows Presentation Foundation (WPF) user controls in Windows Forms applications to provide enhanced rich graphical support within the UI.

- **WPF**. You can use WPF to create a rich client application with UI support for 2-D and 3-D graphics as well as animations and media (both video and audio). WPF also includes a two-way data-binding engine.
- **XAML Browser Application (XBAP) using WPF**. You can create an XBAP that provides all the features of the stand-alone WPF application but is hosted in a browser.

Rich Internet Client (RIA)
The following presentation layer technologies are available for creating rich Internet applications:
- **Silverlight**. You can use Silverlight to create applications that provide a rich user experience that includes graphics, audio, and video.
- **Silverlight with AJAX**. You can combine Silverlight with Asynchronous JavaScript and XML (AJAX) to create an RIA that performs asynchronous communication between the client and the server.

Web Applications
The following technologies are available for creating Web applications:
- **ASP.NET Web Forms**. You can use ASP.NET Web Forms with a wide range of server controls that render HTML in Web browsers.
- **ASP.NET Web Forms with AJAX**. You can use AJAX in your ASP.NET Web Forms application to improve the user experience by reducing the number of postbacks required.
- **ASP.NET Web Forms with Silverlight controls**. You can use Silverlight controls in your ASP.NET Web application to provide a rich user experience and support media streaming.
- **ASP.NET MVC**. You can use ASP.NET MVC to create Web applications with built-in support for the Model-View-Controller (MVC) design pattern. MVC simplifies developing, modifying, and testing the individual components within the application.
- **ASP.NET Dynamic Data**. You can use ASP.NET Dynamic Data to create functional data-driven Web applications based on a LINQ to SQL or Entity Framework data model.

Service Applications
The following technologies are available for creating service applications:
- **Windows Communication Foundation (WCF)**. When possible, use WCF to create services n order to benefit from maximum feature availability and interoperability.
- **ASP.NET Web services (ASMX)**. Use ASMX for simplicity, and only when a suitable Web server will be available.

Whiteboard Your Architecture
It is important that you are able to whiteboard your architecture. Whether or not you share your whiteboard on paper, slides, or other visuals, the key is to show the major constraints and decisions in order to frame and start the conversations.

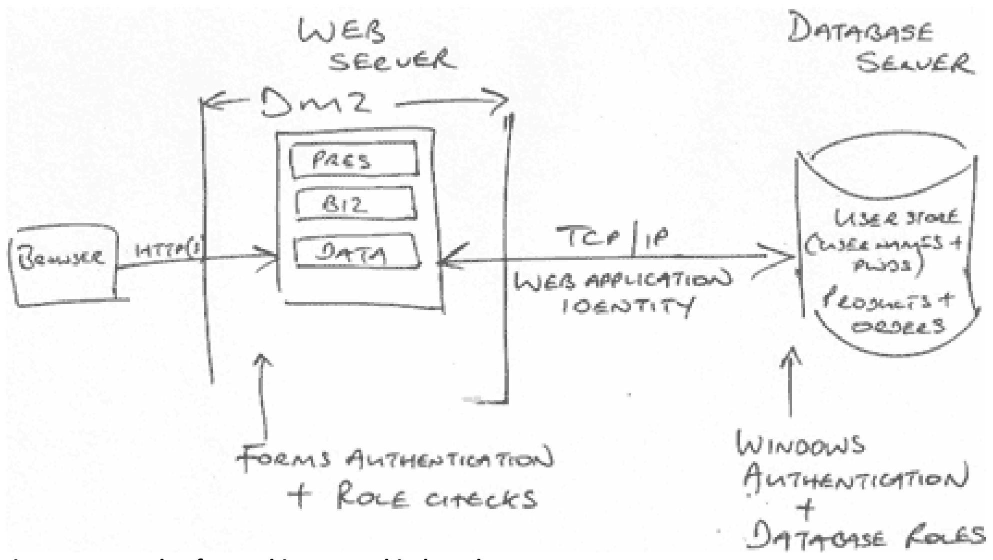
WEB SERVER
DMZ
PRES
BIZ
DATA
BROWSER
HTTP(S)
TCP/IP
WEB APPLICATION IDENTITY
DATABASE SERVER
USER STORE (USER NAMES + PWDS)
PRODUCTS + ORDERS
FORMS AUTHENTICATION + ROLE CHECKS
WINDOWS AUTHENTICATION + DATABASE ROLES

Figure 4 Example of an architecture whiteboard

Step 4: Key Hotspots

Overview

Identify the hotspots in your application architecture to understand the areas where mistakes are most likely to be made. Key hotspots can be organized around quality attributes and cross-cutting concerns.

Architecture Frame

The architecture frame represents cross-cutting concerns that will impact your design across layers and tiers. These are also the areas in which high-impact design mistakes are most often made. Use the architecture frame to identify hotspots in your design that require additional attention to get right.

You can use the following architecture frame to identify cross cutting concerns in your design.

Table 4 Architecture frame

Area	Description
Authentication and Authorization	• How to choose an authentication strategy • How to choose an authorization strategy • How to flow identity across layers and tiers • How to store user identities when not using the Microsoft Active Directory® directory service
Caching and State	• How to choose an appropriate caching technology • How to determine what data to cache • How to determine where to cache the data • How to determine the expiration policy

Area	**Description**
Communication	• How to choose appropriate protocols for communication across layers and tiers • How to design loose coupling across layers • How to perform asynchronous communication • How to pass sensitive data
Composition	• How to choose a composition pattern for the UI) • How to avoid dependencies between modules in the UI • How to handle communication between modules in the UI
Concurrency and Transactions	• How to handle concurrency between threads • How to choose between optimistic and pessimistic concurrency • How to handle distributed transactions • How to handle long-running transactions
Configuration Management	• How to determine what information needs to be configurable • How to determine where and how to store configuration information • How to protect sensitive configuration information • How to handle configuration information in a farm/cluster
Coupling and Cohesion	• How to choose an appropriate layering strategy for separation of concerns • How to design highly cohesive components and group them within layers • How to determine when loose coupling is appropriate between components within a layer
Data Access	• How to manage database connections • How to handle exceptions • How to improve performance • How to handle binary large objects (BLOBs)
Exception Management	• How to handle exceptions • How to log exceptions • How to provide notification when required
Logging and Instrumentation	• How to determine which information to log • How to make the logging configurable • How to determine what level of instrumentation is required
User Experience	• How to improve task efficiency and effectiveness • How to improve responsiveness • How to improve user empowerment • How to improve look and feel
Validation	• How to determine where and how to perform validation • How to validate for length, range, format, and type • How to constrain and reject input • How to sanitize output
Workflow	• How to choose the appropriate workflow technology • How to handle concurrency issues within a workflow • How to handle task failure within a workflow • How to orchestrate processes within a workflow

Quality Attributes

Quality attributes are the cross-cutting concerns that affect run-time performance, system design, and user experience. Quality attributes are important for the overall usability, performance, reliability, and security of software applications. The quality of the application is measured by the extent to which the application possesses a desired combination of these quality attributes. When designing applications to meet any of these requirements, it is necessary to consider the impact on other requirements. You need to analyze the tradeoffs between multiple quality attributes. The importance or priority of each quality attribute differs from system to system; for example, in a line-of-business (LOB) system, performance, scalability, security, and usability will be more important than interoperability, while in a packaged application, interoperability will be very important.

Quality attributes represent areas of concern that have the potential for application-wide impact across layers and tiers. Some attributes are related to the overall system design, while others are specific to run-time, design-time, or user-centric issues. Use the following table to help you organize your thinking about the quality attributes, and to understand which scenarios they are most likely to affect.

Table 5 Quality attributes

Type	Quality attributes
System qualities	• Supportability • Testability
Run-time qualities	• Availability • Interoperability • Manageability • Performance • Reliability • Scalability • Security
Design qualities	• Conceptual Integrity • Flexibility • Maintainability • Reusability
User qualities	• User Experience / Usability

Quality Attribute Frame

The following table describes the quality attributes covered in this chapter. You can use this table to understand what each of the quality attributes means in terms of your application design.

Table 6 Quality attribute frame

Category	Description
Availability	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
Conceptual Integrity	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.
Flexibility	Flexibility is the ability of a system to adapt to varying environments and situations, and to cope with changes to business policies and rules. A flexible system is one that is easy to reconfigure or adapt in response to different user and system requirements.
Interoperability	Interoperability is the ability of diverse components of a system or different systems to operate successfully by exchanging information, often by using services. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
Maintainability	Maintainability is the ability of a system to undergo changes to its components, services, features, and interfaces as may be required when adding or changing the functionality, fixing errors, and meeting new business requirements.
Manageability	Manageability defines how easy it is to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
Performance	Performance is an indication of the responsiveness of a system to execute any action within a given time interval. It can be measured in terms of latency or throughput. *Latency* is the time taken to respond to any event. *Throughput* is the number of events that take place within given amount of time.
Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified time interval.
Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Scalability	Scalability is the ability of a system to function well when there are changes to the load or demand. Typically, the system will be able to be extended over more powerful or more numerous servers as demand and load increase.
Security	Security defines the ways that a system is protected from disclosure or loss of information, and the possibility of a successful malicious attack. A secure system aims to protect assets and prevent unauthorized modification of information.

Category	Description
Supportability	Supportability defines how easy it is for operators, developers, and users to understand and use the application, and how easy it is to resolve errors when the system fails to work correctly.
Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
Usability	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, able to provide good access for disabled users, and able to provide a good overall user experience.

Security Frame

Each quality attribute can be expanded into its own frame to cover the areas where mistakes are most often made. For example, the security frames define a set of pattern-based categories that organize repeatable problems and solutions. You can use these categories to divide your application architecture for further analysis, and to help you identify application vulnerabilities. The categories within the frame represent the critical areas where mistakes are most often made.

Table 7 Security frame

Category	Description / questions
Auditing and Logging	Who did what and when? Auditing and logging refer to how your application records security-related events.
Authentication	Who are you? Authentication is the process where an entity proves the identity of another entity, typically through credentials, such as a username and password.
Authorization	What can you do? Authorization refers to how your application provides access controls for resources and operations.
Configuration Management	Who does your application run as? Which databases does it connect to? How is your application administered? How are these settings protected? Configuration management refers to how your application handles these operations and issues.
Cryptography	How are you handling secrets (confidentiality)? How are you tamper-proofing your data or libraries (integrity)? How are you providing seeds for random values that must be cryptographically strong? Cryptography refers to how your application enforces confidentiality and integrity.
Exception Management	When a method call in your application fails, what does your application do? How much information do you reveal? Do you return friendly information to end users? Do you pass valuable exception information back to the caller? Does your application fail gracefully? Exception management refers to how you handle exceptions within your application.

Category	Description / questions
Input and Data Validation	How do you know that the input your application receives is valid and safe? Input validation refers to how your application filters, scrubs, or rejects input before additional processing. Consider constraining input through entry points and encoding output through exit points. Do you trust data sources such as databases and file shares?
Sensitive data	How does your application handle sensitive data? Does your application protect confidential user and application data? Sensitive data refers to how your application handles any data that must be protected either in memory, over the network, or in persistent stores.
Session Management	How does your application handle and protect user sessions? A session refers to a session of related interactions between a user and your Web application.

You can then use the security frame to determine key security design decisions for your application.

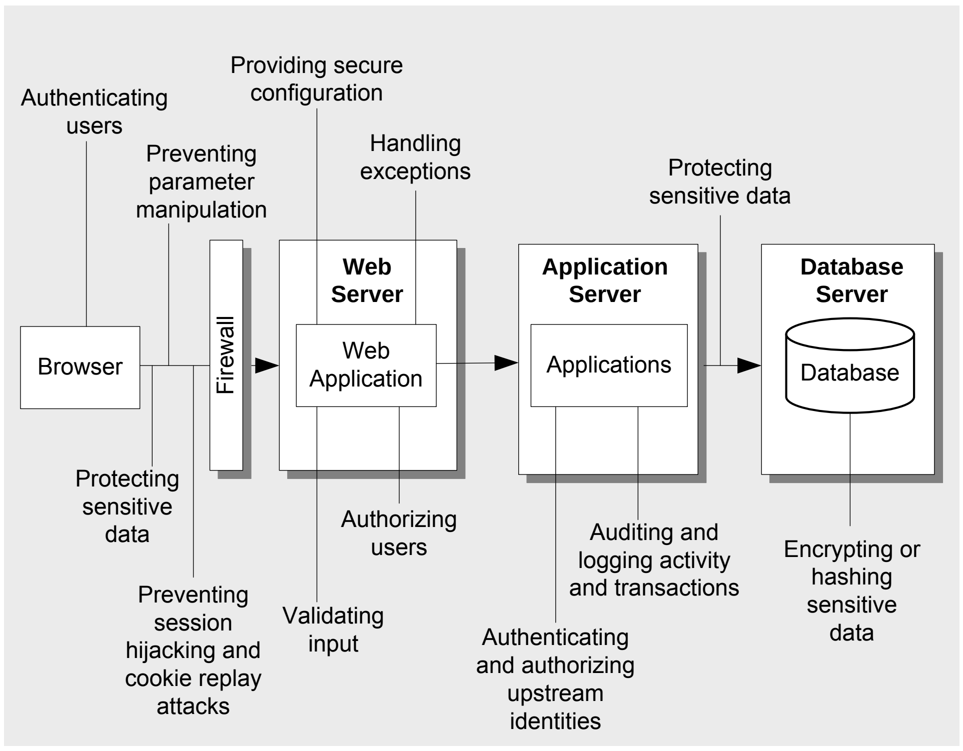
Authenticating users
Providing secure configuration
Preventing parameter manipulation
Handling exceptions
Protecting sensitive data
Browser
Firewall
Web Server
Web Application
Application Server
Applications
Database Server
Database
Protecting sensitive data
Authorizing users
Auditing and logging activity and transactions
Encrypting or hashing sensitive data
Preventing session hijacking and cookie replay attacks
Validating input
Authenticating and authorizing upstream identities

Figure 5 Typical Web application security architecture.

Step 5: Candidate Solutions

After you define the key hotspots, you can create your first high-level design and then start to fill in the details to produce a candidate architecture. You then move back to Step 2 of the process to validate the candidate solution design against the key scenarios and requirements you have defined, before iteratively following the cycle and improving the design.

Architectural spikes are a design prototype you use to determine the feasibility of a specific design path. Use architectural spikes to reduce your risk and quickly determine the viability of different approaches. Test architectural spikes against key scenarios and hotspots.

Consider the following guidelines when creating your architectural spikes:
- Understand your key risks and adapt your design to reduce these risks.
- Optimize your deliverables for effective and efficient communication of design information.
- Build your architecture with flexibility and refactoring in mind. You might need to modify your architecture a number of times, so optimize around this possibility.

What to Do Next
After you complete the architecture-modeling activity, you can begin to refine the design, plan tests, and communicate the design to others. Keep in mind the following guidelines:
- If you capture your candidate architectures and architectural test cases in a document, keep the document lightweight and avoid over-formatting so that you can easily update it. Include the following key content in the document:
    - Your objectives
    - Application type
    - Deployment topology
    - Key scenarios and requirements
    - Technologies
    - Quality attributes
    - Tests
- Use the quality attributes to help shape your design and implementation. For example, developers should be aware of anti-patterns related to the identified architectural risks, and use the patterns to help address the issues.
- Use the architectural frame to plan and scope your architectural tests.
- Communicate the information you capture to relevant team members. This may include your application development team, your test team, and your network and system administrators.

Reviewing Your Architecture
Reviewing the architecture for your application is a critically important task in order to reduce the cost of mistakes and find and fix architectural problems as early as possible. Architecture review is a proven, cost-effective way of reducing project costs and the chances of an architectural failure. Create your architecture to make it as easy as possible to communicate and review. Build your architecture with common review questions in mind, both to improve your architecture and to reduce the time required for each review.

The main goal of an architecture review is to verify that the architecture correctly links the functional requirements and the quality attributes with the proposed technical solution. Additionally, it helps you to identify issues and areas of improvement.

Scenario-based Evaluations

Scenario-based evaluations are a powerful method for reviewing an architecture design. In a scenario-based evaluation, the focus is on the scenarios that are most important from the business perspective, and which have the greatest impact on the architecture. Consider using one of the following common review methodologies:

- **Software Architecture Analysis Method (SAAM)**. SAAM was originally designed for assessing modifiability, but later was extended for reviewing architecture with respect to quality attributes such as modifiability, portability, extensibility, integratibility, and functional coverage. SAAM is also used to review the performance and reliability aspect of the architecture.
- **Architecture Tradeoff Analysis Method (ATAM)**. ATAM is a refined and improved version of SAAM that helps you review architectural decisions with respect to the quality attributes requirements, and how well they satisfy particular quality goals.
- **Active Design Review (ADR)**. This architecture review technique is best suited for incomplete or in-progress architectures. The main difference is that the review is more focused on a set of issues or individual sections of the architecture at a time, rather than performing a general review.
- **Active Reviews of Intermediate Designs (ARID)**. This architecture review technique combines the ADR aspect of reviewing in-progress architecture with a focus on a set of issues, and the ATAM and SAAM approach of scenario-based review focused on quality attributes.
- **Cost Benefit Analysis Method (CBAM)**. This architecture review technique focuses on analyzing the costs, benefits, and schedule implications of architectural decisions.
- **Architecture Level Modifiability Analysis (ALMA)**. This architecture review technique evaluates the modifiability of architecture for business information systems (BIS).
- **Family Architecture Assessment Method (FAAM)**. This architecture review technique evaluates information system family architectures for interoperability and extensibility.

Communicating Your Architecture Design

After your architecture design is complete, you must communicate the design to the other stakeholders. These stakeholders will include the development team, system administrators and operators, business owners, and other interested parties. There are several well-known methods for describing architecture to others, including the following:

- **4+1**. This approach uses five views of the complete architecture. Four of the views describe the architecture from different approaches: the logical view (such as the object model), the process view (such as concurrency and synchronization aspects), the physical view (the map of the software layers and functions onto the distributed hardware infrastructure), and the development view. A fifth view shows the scenarios and use cases for the software. This allows stakeholders to see the aspects of the architecture that specifically interest them.
- **Architecture Description Language (ADL)**. This approach is used to describe software architecture prior to system implementation. It addresses the following concerns: behavior, protocol, and connector. Behavior corresponds to the types, hierarchies, properties, and rules. Protocol corresponds to the communicating entities and rules. Connector

corresponds to the connections, interface, and constraints that exist between components in an object-oriented system. ADL is designed to be both human- and machine-readable. Therefore, ADL has both the textual form and a formally defined syntactic and semantic form. The main advantage of ADL is that you can analyze the architecture for completeness, consistency, ambiguity, and performance before formally beginning use of the design.

- **Agile Modeling**. This approach follows the concept that "content is more important than representation." This ensures that the models created are simple and easy to understand, sufficiently accurate, detailed, and consistent. Agile model documents target specific customer(s) and fulfill the work efforts of that customer. Each agile model document is designed to fulfill a specific purpose, and the way it is expressed can vary. A class diagram, problem statement, use case diagram, sequence flow, and other approaches can be used to express an agile model document. The simplicity of the document ensures that there is active participation of stakeholders in the modeling of the artifact.
- **IEEE 1471**. IEEE 1471 is the short name for a standard formally known as ANSI/IEEE 1471-2000, "Recommended Practice for Architecture Description of Software-Intensive Systems." IEEE 1471 enhances the content of an architectural description, in particular giving specific meaning to context, views, and viewpoints. Context corresponds to the stakeholders of the system (such as clients and vendors) and their specific concerns (such as reliability and security). The view corresponds to the system concerns, and the viewpoint corresponds to conditions on the completeness, well-formedness, and analyzability of views. IEEE 1471 allows the document to be formulated by using existing ADLs, or from within the framework it provides. IEEE 1471 also allows UML diagrams (see below) to be used as viewpoints.
- **Unified Modeling Language (UML)**. This approach represents three views of a system model. The functional requirements view (functional requirements of the system from the point of view of the user, including use cases); the static structural view (objects, attributes, relationships, and operations including class diagrams); and the dynamic behavior view (collaboration among objects and changes to the internal state of objects, including includes sequence, activity, and state diagrams).

Chapter 5: Deployment Patterns

Objectives

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

Overview

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

This chapter describes the options available for deployment of different types of 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 because of technical infrastructure limitations.

Choosing a Deployment Strategy

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

When choosing a deployment strategy:

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

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

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

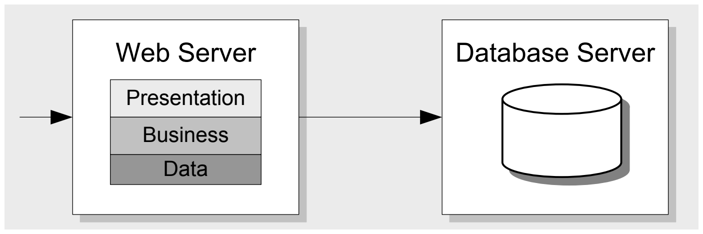

Figure 1 Non-distributed deployment

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

Distributed Deployment

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

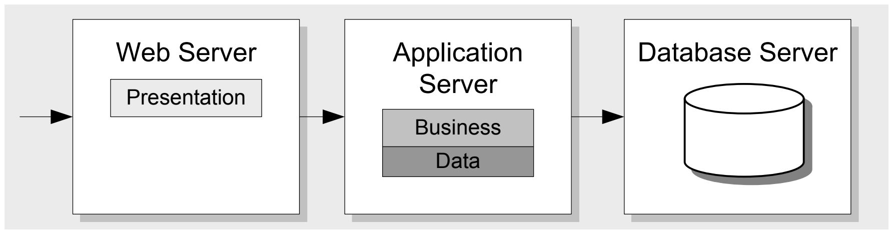

Figure 2 Distributed deployment

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

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

Performance and Design Considerations for Distributed Environments

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

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

- Choose communication paths and protocols between tiers to ensure that components can securely interact with minimum performance degradation.
- Consider using services and operating system features such as distributed transaction support and authentication that can simplify your design and improve interoperability.
- Reduce the complexity of your component interfaces. Highly granular interfaces ("chatty" interfaces) that require many calls to perform a task work best when located on the same physical machine. Interfaces that make only one call to accomplish each task ("chunky" interfaces) provide the best performance when the components are distributed across separate physical machines.
- Consider separating long-running critical processes from other processes that might fail by using a separate physical cluster.
- Determine your failover strategy. For example, Web servers typically provide plenty of memory and processing power, but may not have robust storage capabilities (such as RAID mirroring) that can be replaced rapidly in the event of a hardware failure.
- Take advantage of asynchronous calls, one-way calls, or message queuing to minimize blocking when making calls across physical boundaries.
- Determine how best to plan for the addition of extra servers or resources that will increase performance and availability.

Recommendations for Locating Components within a Distributed Deployment

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

Consider the following guidelines when determining where to locate components in a distributed environment:

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

Scale Up vs. Scale Out

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

When you scale your application, you can choose from and combine two basic choices:
- Scale up: Get a bigger box.
- Scale out: Get more boxes.

Scale Up: Get a Bigger Box

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

Scale Out: Get More Boxes

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

Consider Whether You Need to Support Scale Out

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

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

- **You need to be able to scale out your bottlenecks, wherever they are**. If the bottlenecks are on a shared resource that cannot be scaled, you have a problem. However, having a class of servers that have affinity with one resource type could be beneficial, but they must then be independently scaled. For example, if you have a single Microsoft SQL Server® instance that provides a directory, everyone uses it. In this case, when the server becomes a bottleneck, you can scale out and use multiple copies. Creating an affinity between the data in the directory and the SQL Servers that serve the data allows you to concentrate those servers and does not cause scaling problems later, so in this case affinity is a good idea.
- **Define a loosely coupled and layered design**. A loosely coupled, layered design with clean, remotable interfaces is more easily scaled out than tightly-coupled layers with "chatty" interactions. A layered design will have natural clutch points, making it ideal for scaling out at the layer boundaries. The trick is to find the right boundaries. For example, business logic may be more easily relocated to a load-balanced, middle-tier application server farm.

Consider Design Implications and Tradeoffs up Front

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

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

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

- **Static, reference, and read-only data**. For this type of data, you can easily have many replicas in the right places if this helps your performance and scalability. This has minimal impact on design and can be largely driven by optimization considerations. Consolidating several logically separate and independent databases on one database server may or may not be appropriate even if you can do it in terms of capacity. Spreading replicas closer to the consumers of that data may be an equally valid approach. However, be aware that whenever you replicate, you will have a loosely synchronized system.
- **Dynamic (often transient) data that is easily partitioned**. This is data that is relevant to a particular user or session (and if subsequent requests can come to different Web or application servers, they all need to access it), but the data for user A is not related in any way to the data for user B. For example, shopping carts and session state both fall into this category. This data is slightly more complicated to handle than static, read-only data, but you can still optimize and distribute quite easily. This is because this type of data can be partitioned. There are no dependencies between the groups, down to the individual user level. The important aspect of this data is that you do not query it across partitions. For example, you ask for the contents of user A's shopping cart but do not ask to show all carts that contain a particular item.
- **Core data**. This type of data is well maintained and protected. This is the main case where the "scale up, then out" approach usually applies. Generally, you do not want to hold this type of data in many places because of the complexity of keeping it synchronized. This is the classic case in which you would typically want to scale up as far as you can (ideally, remaining a single logical instance, with proper clustering), and only when this is not enough, consider partitioning and distribution scale-out. Advances in database technology (such as distributed partitioned views) have made partitioning much easier, although you should do so only if you need to. This is rarely because the database is too big, but more often it is driven by other considerations such as who owns the data, geographic distribution, proximity to the consumers, and availability.

Consider Database Partitioning at Design Time

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

Partitioning provides several benefits:

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

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

Performance Patterns

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

Web Farms

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

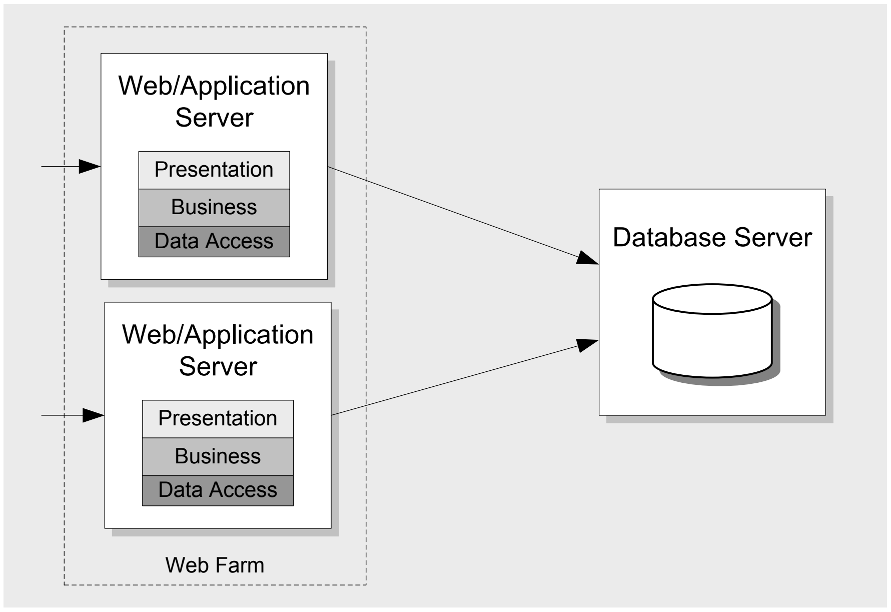

Figure 3 A simple Web farm

Affinity and User Sessions

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

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

Application Farms

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

Load-balancing Cluster

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

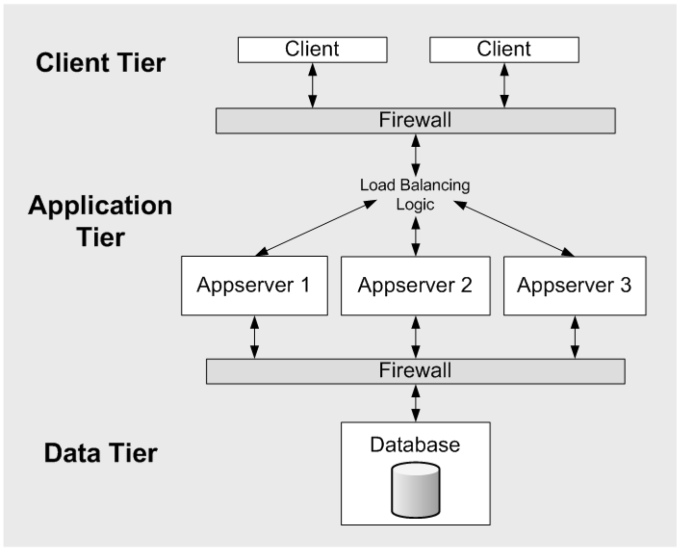

Figure 4 A load-balanced cluster

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

Reliability Patterns

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

Failover Cluster

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

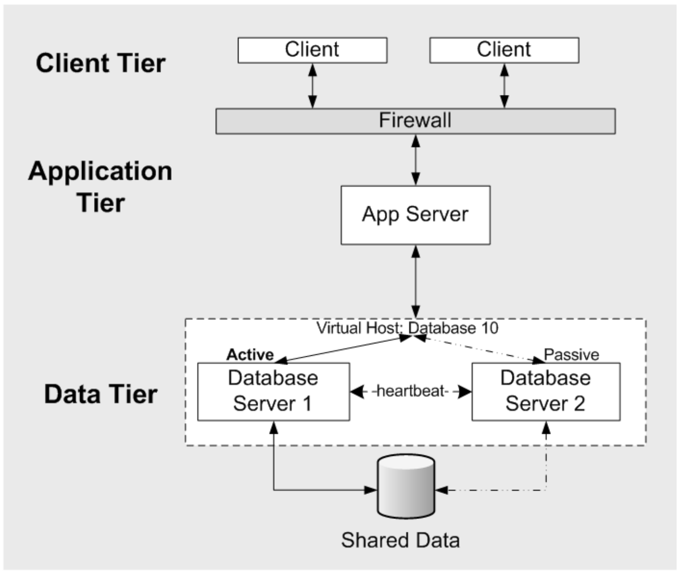

Figure 5 A failover cluster

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

Security Patterns

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

Impersonation/Delegation

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

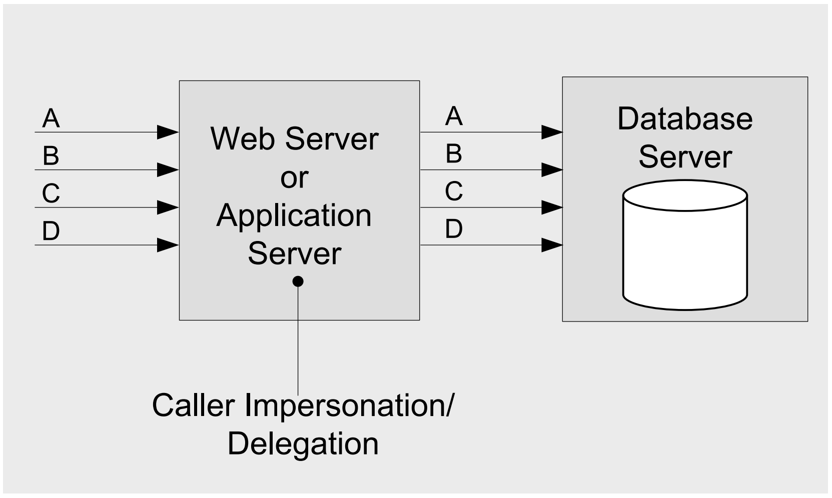
A
B
C
D
Web Server or Application Server
A
B
C
D
Database Server
Caller Impersonation/ Delegation

Figure 6 The impersonation/delegation authorization model

Trusted Subsystem

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

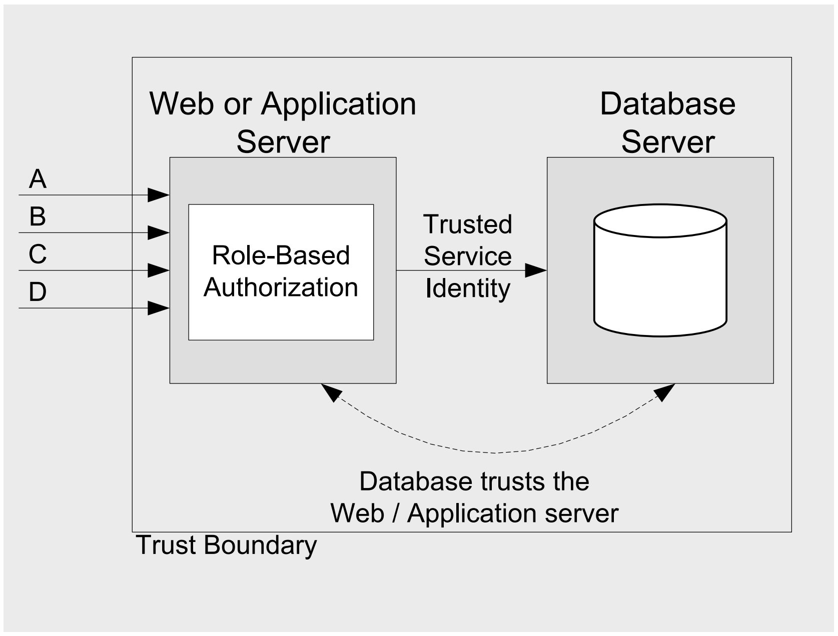

Figure 7 The trusted subsystem (or trusted server) model

Multiple Trusted Service Identities

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

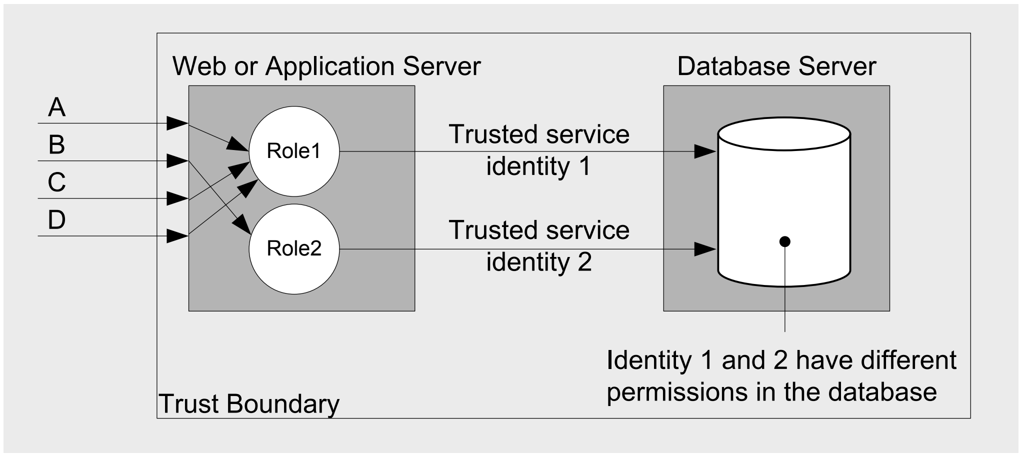
Web or Application Server
Database Server
A
B
C
D
Role1
Role2
Trusted service identity 1
Trusted service identity 2
Identity 1 and 2 have different permissions in the database
Trust Boundary

Figure 8 The multiple trusted service identities model

Network Infrastructure Security Considerations

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

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

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

Manageability Considerations

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

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

Pattern Map

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

Key Patterns

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

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.

Chapter 6: Architectural Patterns and Styles

Objectives

- Learn key architectural patterns and styles.
- Learn about the scenarios, key principles, and benefits of each style.
- Learn how to choose an appropriate architectural style for your application.

Overview

This chapter describes and discusses the architectural styles commonly used for applications today. These styles include client/server, layered architecture, component-based architecture, message-bus architecture, and service-oriented architecture (SOA). For each style, you will find an overview, key principles, major benefits, and examples of its use. It is important to understand that the styles describe different aspects of applications. For example, some architectural styles describe deployment patterns, some describe structure and design factors, and others describe communication factors. Therefore, a typical application will usually implement more than one of the styles described in this chapter.

What Is an Architectural Style?

An architectural style is a set of principles. You can think of it as a coarse-grained pattern that provides an abstract framework for a family of systems. Each style defines a set of rules that specify the kinds of components you can use to assemble a system, the kinds of relationships used in their assembly, constraints on the way they are assembled, and assumptions about the meaning of how you put them together. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems.

You can think of architecture styles as sets of principles that shape an application. An architectural pattern, or as it is sometimes called, an architectural style was defined by Garlan and Shaw as:

> "…a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition." [*David Garlan and Mary Shaw, January 1994, CMU-CS-94-166*]

Architecture Style Frame

Architectural styles can be organized by their key focus area. The following table lists the major areas of focus and the corresponding architectural styles.

Table 1 Architectural style frame

Category	Architecture styles
Deployment	Client/server, 3-Tier, N-Tier
Structure	Component-Based, Object-Oriented, Layered Architecture, Model-View-Controller (MVC)
Domain	Domain Model, Gateway
Communication	Service-Oriented Architecture (SOA), Message Bus, Pipes and Filters

These styles are not exclusive, and you will often choose multiple overlapping styles to suit your architectural needs. For example, you might design with object-oriented code principles, organize as a layered architecture, use the Domain model for data access, communicate with services using a service-oriented architecture (SOA), and deploy using an N-tier style.

Key Architecture Styles and Patterns

The following table lists the common architectural styles and patterns described in this chapter. It also contains a brief description of each style or pattern. Later sections of this chapter contain more details of each style, as well as guidance to help you choose the appropriate ones for your application.

Table 2 Architectural styles and descriptions

Architecture style	Description
Client-Server	Segregates the system into two applications, where the client makes a service request to the server.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that are location-transparent and expose well-defined communication interfaces.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message-Bus	A software system that can receive and send messages that are based on a set of known formats, so that systems can communicate with each other without needing to know the actual recipient.
Model-View-Controller (MVC)	Separates the logic for managing user interaction from the user interface (UI) view and from the data with which the user works.
N-tier / 3-tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	An architectural style based on division of tasks for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to Applications that expose and consume functionality as a service using contracts and messages.

Architecture Style Matrix

Many applications will use a combination of architectural styles as part of the complete system. For example, you might have an SOA design composed of services developed using a layered architecture approach and object oriented design. The following matrix provides guidance on how architectural styles may be combined within a solution. Note that you should interpret the matrix by reading from left to right, row by row. For example, if you are building an application that follows the layered style, it is *possible* to implement it using SOA architecture. However, when you are building an application based on the SOA style, it is *recommended* that you use layered architecture.

Table 3 Architecture style matrix

	Client/Server	**Component-based**	**Layered**	**Message bus**	**MVC**	**N-tier**	**OO**	**SOA**
Client/server		R	R	P	U	U	R	P
Component-based	P		P	P	U	P	R	P
Layered	P	R		P	P	P	R	P
Message-bus	P	R	R		U	P	R	P
MVC	U	P	P	U		U	R	U
N-tier	U	R	R	P	U		R	P
OO	P	P	P	P	P	P		P
SOA	P	R	R	P	U	P	R	

Key: P=Possible, U=Unlikely, R=Recommended

The following are two examples of using the matrix.

When choosing the layered style:
- It is possible to also use the client/server style.
- It is recommended to also use the component-based style.
- It is possible to also use the message-bus style.
- It is possible to also use the MVC style.
- It is possible to also use the N-tier style.
- It is recommended to also use the OO style.
- It is possible to also use the SOA style.

When choosing the SOA style:
- It is possible to also use the client/server style.
- It is recommended to also use the component-based style.
- It is recommended to also use the layered style.
- It is possible to also use the message-bus style.
- It is unlikely to also use the MVC style.

- It is possible to also use the N-tier style.
- It is recommended to also use the OO style.

Choosing an Architectural Style

Many factors will influence the architectural styles that you choose. These factors include the capacity of your organization for design and implementation; the capabilities and experience of developers; and the hardware and deployment scenarios available. The architecture styles are related, so effectively you will be choosing a mix of architecture for your scenario. For example, when choosing a layered architecture, you may also choose to use an object-oriented or component-based design. The following guidelines will help you to determine the appropriate styles for your applications.

3-Tier/N-Tier Architectural Style

Consider either the N-tier or the 3-tier architectural style if:

- The processing requirements of the layers in the application differ. Processing in one layer could absorb sufficient resources to slow the processing in other layers.
- The security requirements of the layers in the application may differ. For example, the presentation layer will not store sensitive data, while this may be stored in the business and data layers.
- You want to be able to share business logic between applications.
- You have sufficient hardware to allocate the required number of servers to each tier.

Consider the 3-tier architectural style if:

- You are developing an intranet application where all servers are located within the private network.
- You are developing an Internet application, and security requirements do not restrict implementing business logic within the public-facing Web or application server.

Consider using more than three tiers if:

- Security requirements dictate that business logic cannot be deployed to the perimeter network.
- The application makes heavy use of resources and you want to offload that functionality to another server

Client/Server Architectural Style

Consider the client/server architectural style if:

- Your application is server-based and will support many clients.
- You are creating Web-based applications exposed through a Web browser.
- You are implementing business processes that will be used by people throughout the organization.
- You are creating services for other applications to consume.
- You want to centralize data storage, backup, and management functions.

- Your application must support different client types and different devices.

Component-based Architectural Style

Consider the component-based architectural style if:

- You already have suitable components, or can obtain suitable components from third-party suppliers.
- Your application will predominantly execute procedural-style functions, perhaps with little or no data input.
- Your application is relatively simple, and does not warrant a full layered architecture.
- Your application has specific requirements that do not include a UI or business processes.
- You want to be able to combine components written in different code languages.
- You want to create a pluggable architecture that allows you to easily replace and update individual components.

Layered Architectural Style

Consider the layered architectural style if:

- You already have suitable layers built for other applications that you can reuse.
- You already have applications that expose suitable business processes through service interfaces.
- Your application is complex, and the high-level design demands separation so that teams can focus on different areas of functionality.
- Your application must support different client types and different devices.
- You want to implement complex and/or configurable business rules and processes.

Message-bus Architectural Style

Consider the message-bus architectural style if:

- You have existing applications that interoperate with each other to perform tasks.
- You are implementing a task that requires interaction with external applications.
- You are implementing a task that requires interaction with applications hosted in different environments.
- You have existing applications that perform specific tasks, and you want to combine those tasks into a single operation.

Model-View-Controller (MVC) Architectural Style

Consider the Model-View-Controller architectural style if:

- You want improved testability and simpler maintenance of UI functionality.
- You want to separate the task of creating the UI from the logic code that drives it.
- Your UI view does not contain any request-processing code.
- Your UI processing code does not implement any business logic.

Object-Oriented Architectural Style

Consider the object-oriented architectural style if:

- You want to model the application based on real-world objects and actions.
- You already have suitable objects and classes that match the design and operational requirements.
- You need to encapsulate logic and data together in reusable components.
- You have complex business logic that requires abstraction and dynamic behavior.
- You have the capacity and resources to perform an in-depth analysis of the business domain.
- You are building an application where the return on investment (ROI) outweighs the initial cost of analysis and design.

Service-Oriented Architectural (SOA) Style

Consider the SOA style if:

- You have access to suitable services, or can purchase suitable services exposed by a hosting company.
- You want to build applications that compose a variety of services into a single UI.
- You are creating Software plus Services (S+S), Software as a Service (SaaS), or cloud-based applications.
- You need to support message-based communication between segments of the application.
- You need to expose functionality in a platform-independent way.
- You want to take advantage of federated services, such as authentication.
- You want to expose services that are discoverable through directories and can be used by clients that have no prior knowledge of the interfaces.

Architecture Style Summaries

The following are summaries of each of the architecture styles listed earlier in this chapter. Each summary includes a brief description of the architecture style along with key principles, benefits, and some examples.

Client/Server Architecture

Client/server describes the relationship between two computer programs in which one program, the client, makes a service request from another program, the server. Standard networked functions such as e-mail exchange, Web access, and database access are based on the client/server model.

The client/server architectural style has the following identifying characteristics:

- It is a style for defining distributed systems.
- It involves a separate client and server system, and a connecting network.
- It describes the relationship between the client and one or more servers, with the client making requests to the server, and the server sending responses.
- It can use a range of protocols and data formats to communicate information.

Key Principles

The following are the key principles of the client/server architectural style:

- The client initiates one or more requests, waits for replies, and processes the replies on receipt.
- The client usually connects to one or only a small number of servers at any one time.
- The client interacts directly with the user, perhaps using a graphical UI.
- The server does not initiate any requests.
- The server sends data responses to network requests from connected clients.
- The server typically authorizes the user and then carries out the processing required to generate the result.

Benefits

The main benefits of the client/server architectural style are:

- **Higher security**. All data is stored on the server, which generally have offers greater control of security than clients do.
- **Centralized data access**. Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- **Ease of maintenance**. Roles and responsibilities of a computing system are distributed among several servers known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

Examples

The following are some examples of the client/server architectural style:

- Web browser–based programs running on the Internet or an intranet
- Windows Forms applications that access back-end services
- Applications that access remote data stores, such as e-mail readers, File Transfer Protocol (FTP) clients, and database query tools
- Tools and utilities that manipulate remote systems, such as system management tools and network monitoring tools

Variations on the client/server style include:

- **Client-Queue-Client systems**. Clients can read data from and send data to a server that acts simply as a queue to store the data. This allows clients to distribute and synchronize files and information.
- **Peer-to-Peer (P2P) applications**. Developed from the Client-Queue-Client style, the P2P style allows the client and server to swap their roles in order to distribute and synchronize files and information across multiple clients.
- **Application servers**. A specialized client/server architectural style where the server hosts and executes applications that a thin client controls through specialized client-installed software.

Component-Based Architecture

Component-based architecture describes a software engineering approach to system design and development. It focuses on the decomposition of the design into functional or logical components that expose well-defined communication interfaces. This provides a higher level of abstraction than object-oriented design principles, and does not focus on object-specific issues such as communication protocols and shared state.

The component-based architectural style has the following identifying characteristics:

- It is a style for designing applications made up of individual components.
- It places the emphasis on decomposing systems into functional or logical components that have well-defined interfaces.
- It defines a design approach that uses discrete components, which communicate through interfaces containing methods, events, and properties.

Key Principles

A *component* is a software object specifically designed to achieve a certain purpose. The key principles when designing components are that they should be:

- **Reusable**. Components are usually designed to be reused in different scenarios in different applications. However, some components may be designed for a specific task.
- **Not context-specific**. Components are designed to operate in different environments and contexts. Specific information, such as state data, should be passed to the component instead of being included in or accessed by the component.
- **Extensible**. A component can be extended from existing components to provide new behavior.
- **Encapsulated**. Components expose interfaces that allow code to use its functionality, and not reveal internal details of the processes or any internal variables or state.
- **Independent**. Components are designed to have minimal dependencies on other components. Therefore components can be deployed into any appropriate environment without affecting other components or systems.

Benefits

The following are the main benefits of the component-based architectural style:

- **Ease of deployment**. As new compatible versions become available, you can replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost**. The use of third-party components allows you to spread the cost of development and maintenance.
- **Ease of development**. Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- **Reusable**. The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.

- **Mitigation of technical complexity**. Components mitigate complexity through the use of a component container and its services. Example component services include component activation, lifetime management, method queuing, eventing, and transactions.

Examples
Common types of components used in applications include:
- User interface components, such as grids and buttons, often referred to as “controls.”
- Helper components that expose a specific subset of functions used in other components.
- Components that are infrequency accessed, resource-intensive, and must be activated using the Just in Time (JIT) approach. This is common in remoting or distributed component scenarios.
- Queued components, whose method calls may be executed asynchronously using message queuing and store-and-forward.

Layered Architecture
Layered architecture focuses on a hierarchical distribution of roles and responsibilities by providing a highly effective separation of concerns. The role indicates the mode and type of interaction with other layers, and the responsibility indicates the functionality being addressed. For example, a typical Web application design comprises a presentation layer (functionality related to the UI), a Business Layer (business rules processing), and a data layer (functionality related to data access).

The layered architectural style has the following identifying characteristics:
- It describes decomposition of services so that the majority of interactions occur only between neighboring layers.
- The layers of an application may reside on the same physical computer (the same tier), or may be distributed over separate computers (N-tier).
- The components in each layer communicate with components in other layers through well-defined interfaces.
- It has been described as an “inverted pyramid of reuse” where each layer aggregates the responsibilities and abstractions of the layer directly beneath it.

Key Principles
Common principles to apply when designing to use this style or architecture include:
- **Abstraction**. Layered architecture abstracts the view of the model as whole while providing enough detail to understand the relationship between layers.
- **Encapsulation**. The design does not make assumptions about data types, methods and properties, and implementation.
- **Clearly defined functional layers**. The design clearly defines the separation between functionality in each layer. Upper layers such as the presentation layer send commands to lower layers such as the business and data layers, and data flows both up and down between the layers.
- **High cohesion**. Each layer contains functionality directly related to the tasks of that layer.

- **Reusable**. Lower layers have no dependencies on higher layers, allowing them to be reusable in other scenarios.
- **Loose coupling**. Communication between layers is based on abstraction that provides loose coupling between layers.

Benefits

The main benefits of the layered architectural style are:

- **Abstraction**. Layers allow changes to be made at the abstract level. You can increase or decrease the level of abstraction you use in each layer of the hierarchical "stack."
- **Isolation**. The layered architectural style allows you to isolate technology upgrades to certain layers in order to reduce risk and impact to the overall system.
- **Performance**. Distributing the layers over multiple physical tiers can improve scalability, fault-tolerance, and performance.
- **Testability**. Testability benefits from having well-defined layer interfaces as well as the ability to switch between different implementations of the layer interfaces.
- **Independent**. The layered architectural style removes the requirement to consider hardware and deployment issues and external interface dependencies.

Examples

Common types of layered applications include:

- Line of business (LOB) applications, such as accounting and customer-management systems.
- Enterprise Web-based applications and Web sites.
- Enterprise desktop or smart clients with centralized application servers for business logic.

The following are some variations on the layered architectural style:

- **Strict layering**. Each layer is only allowed to call the layer directly below it.
- **Layer skipping**. Layers are allowed to call layers deeper than the one directly below them. This is known as *layer skipping*, and can increase performance but will impact portability.
- **Black-box layering**. Layer boundaries and dependencies are strictly defined using interfaces, which supports run-time extension, interception, and improved testability.
- **White-box layering**. Classes collaborating across layer boundaries are tightly coupled.

Message-bus Architecture

Message-bus architecture describes the principle of using a software system that can receive and send messages using one or more communication channels, so that applications can interact without having to know specific details about each other. The most common implementations of message-bus architecture use either a messaging router or a Publish/Subscribe pattern.

The message-bus architectural style has the following identifying characteristics:

- It is a style for designing applications where interaction between applications is accomplished by passing messages over a common bus.

- Communication between applications using message-bus architecture is normally asynchronous.
- It is often implemented using a messaging system, such as Microsoft Message Queuing (MSMQ).
- Many implementations consist of individual applications that communicate using common schemas and a shared infrastructure for sending and receiving messages.

Key Principles
A message-bus provides the ability to handle:
- **Message-oriented communications**. All communication between applications is based on messages that use known schemas.
- **Complex processing logic**. Complex operations can be created by combining a set of smaller operations supporting specific tasks.
- **Modifications to processing logic**. Because interaction with the bus is based on common schemas and commands, you can insert or remove applications on the bus to change the logic used to process messages.
- **Integration with different environments**. By using a message-based communication model based on common standards, you can interact with applications developed for different environments, such as Microsoft .NET and Java.

Benefits
The main benefits of the message-based architectural style are:
- **Expandability**. Applications can be added to or removed from the bus without having an impact on the existing applications.
- **Low complexity**. Application complexity is reduced because each application only needs to know how to communicate with the bus.
- **Higher performance**. Performance is higher due to the absence of intermediaries between communicating applications, limited only by how fast the message bus can move messages.
- **Scalability**. Multiple instances of the same application can be attached to a bus in order to handle multiple requests at the same time.
- **Simplicity**. Each application needs to support only a single connection to the message bus instead of multiple connections to other applications.

Examples
Message-bus designs have been used to support complex processing rules for many years. The design provides a pluggable architecture that allows you to insert applications into the process, or improve scalability by attaching several instances of the same application to the bus.

Variations on the message bus style include:
- **Enterprise service bus (ESB)**. Based on message-bus designs, an ESB uses services for communication between the bus and components attached to the bus. An ESB will usually provide services that transform messages from one format to another, allowing clients that use incompatible message formats to communicate with each other

- **Internet service bus (ISB)**. This is similar to an enterprise service bus, but with applications hosted in the cloud instead of on an enterprise network. A core concept around ISB is the use of Uniform Resource Identifiers (URI) and policies to control the routing of logic through applications and services in the cloud.

Model-View-Controller (MVC)

Model-View-Controller (MVC) represents a style for handling requests or user actions, and for manipulating the UI and the application data. The MVC architecture separates UI processing into three specific roles called the Model, the View, and the Controller. The Model represents data, the View provides a UI, and the Controller provides processing logic.

An MVC architectural style has the following identifying characteristics:
- It is a style for designing applications based on well-known design patterns.
- It separates the logic for managing user interaction from the UI view, and from the data with which the user works.
- It allows graphical designers to create a UI, while developers generate the code to drive it.
- It provides support for improved testability. Dividing the functionality into separate roles provides increased opportunities to test the behavior of individual roles.

Key Principles

The following are the key principles of the MVC architectural style:
- **Separation of concerns**. The MVC architectural style separates UI processing concerns into three separate roles: Model, View, and Controller. The Model represents data, the View represents a UI, and the Controller is used to handle requests and perform operations.
- **Improved testability**. The Passive Model MVC pattern allows you to build mock objects that mimic the behavior of concrete objects during testing.
- **Event-based notification**. The Observer pattern is commonly used to provide notifications to the View when data managed by the Model changes.
- **Delegated event handling**. The controller handles events triggered from the UI controls in the view.

Benefits

The main benefits of the MVC/MVP architectural style are:
- **Testability**. Because the Model and Controller are only classes, they can be unit-tested in the same way as any other class. You can also replace one of these classes with mock objects that provide simulated behavior.
- **Reusability**. The Controller can be reused with other compatible Views, and a View can be reused with other compatible Controllers.
- **Manageability**. Separation of core concerns helps to identify dependencies and organizes the code into more manageable sections.

Examples
Some examples of the MVC architectural style are:
- ASP.NET MVC
- Menu Interaction in Windows Forms

N-Tier / 3-Tier Architecture
This architectural deployment style describes the separation of functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer. It evolved through the component-oriented approach, generally using platform-specific methods for communication instead of a message-based approach.

Key Principles
The following are the key principles of the N-tier/3-tier architectural style:
- It is a style for defining the deployment of the layers of an application.
- N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization.
- Each tier is completely independent of all other tiers, except for those immediately above and below it. The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th tier (if there is one), and how to handle the results of the request.
- N-tier architectures have at least three separate logical layers, or parts. Each layer has specific functionality that it is responsible for, and located on, physically different servers.
- A layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.

Benefits
The main benefits of the N-tier/3-tier architectural style are:
- **Maintainability**. Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- **Scalability**. Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- **Flexibility**. Because each tier can be managed or scaled independently, flexibility is increased.
- **Availability**. Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Examples
Some examples of the N-tier/3-tier architectural style are:

- A typical financial Web application, where security is important and the business layer needs to be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter.
- A typical rich client connected application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on the server.

Object-oriented Architecture

Object-oriented architecture is a programming style based on the division of tasks for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object. An object-oriented design views a system as a series of cooperating objects, instead of a set of routines or procedural instructions. Objects are discrete, independent, and loosely coupled; they communicate through interfaces, and by sending and receiving messages.

An object-oriented architectural style has the following identifying characteristics:

- It is a style for designing applications based on discrete units of logical and reusable code.
- It describes the use of self-sufficient objects that contain data and the behavior to work with that data, and have a distinct role or responsibility.
- It emphasizes reusability through encapsulation, modularity, polymorphism, and inheritance.
- It contrasts with the procedural approach, where there is a predefined sequence of tasks and actions. The object-oriented approach uses the concept of objects interacting with each other to perform a range of tasks.

Key Principles

The key principles of the object-oriented architectural style are:

- **Abstraction**. This allows you to reduce a complex operation into a generalization that retains the base characteristics of the operation. For example, an abstract interface can be a well-known definition that supports data access operations using simple methods such as **Get** and **Update**. Another form of abstraction could be metadata used to provide a mapping between two formats that hold structured data.
- **Composition**. Objects can be assembled from other objects, and can choose to hide these “internal” objects from other classes or expose them as simple interfaces.
- **Inheritance**. Objects can inherit from other objects, and use functionality in the base object or override it to implement new behavior. Moreover, inheritance makes maintenance and updates easier, as changes to the base object are propagated automatically to the inheriting objects.
- **Encapsulation**. Objects expose only the functionality through methods, properties, and events, and hide the internal details such as state and variables to other objects. This makes it easier to update or replace objects, providing that their interfaces are compatible, without affecting other objects and code.
- **Polymorphism**. This allows you to override the behavior of a base type that supports operations in your application.

- **Decoupling**. This allows objects to be decoupled from the consumer by defining an abstract interface that the object implements and the consumer know about. This allows you to provide alternative implementations without affecting consumers of the interface.

Benefits

The main benefits of the object-oriented architectural style are:

- **Understandable**. Object-oriented design maps the application more closely to the real-world objects, making it more understandable.
- **Reusable**. Object-oriented design provides for reusability through polymorphism and abstraction.
- **Testable**. Object-oriented design provides for improved testability through encapsulation.
- **Extensible**. Encapsulation, polymorphism, and abstraction ensure that change in the representation of data does not affect the interfaces that it exposes.

Examples

Common uses of the object-oriented style include:

- Defining objects that represent real-world artifacts within a business domain, such as a customer or an order.
- Defining an object model that supports complex scientific or financial operations.

Service-Oriented Architecture (SOA)

Service-oriented architecture enables application functionality to be provided and consumed as a set of services. Services use a standards-based form of interface that can be invoked, published, and discovered. SOA services are focused on providing a schema and message-based interaction with an application. SOA services provide application-scoped interfaces and not component or object-based interfaces. In other words, a SOA service should not be treated as a component-based service provider.

The SOA style has the following identifying characteristics:

- Interaction with a service is loosely-coupled.
- It can involve business processes packaged into interoperable services.
- Clients and other services can access local services running on the same tier.
- Clients and other services access remote services over a connecting network.
- It can use a range of protocols and data formats to communicate information.

Key Principles

The key principles of the SOA architectural style are:

- **Services are autonomous**. Each SOA service is maintained, developed, deployed, and versioned independently.
- **Services are distributable**. SOA services can be located anywhere on a network, locally or remotely, as long as the network supports the required communication protocols.

- **Services are loosely-coupled**. Each SOA service is independent of others, and can be replaced or updated without breaking applications that use it as long as the interface is still compatible.
- **Services share schema and contract, not class**. SOA services share contracts and schemas when they communicate, not internal classes.
- **Compatibility is based on policy**. Policy in this case means definition of features such as transport, protocol, and security.

Benefits
The main benefits of the SOA architectural style are:
- **Domain alignment**. Reuse of common services with standard interfaces increases business and technology opportunities and reduces cost.
- **Abstraction**. Services are autonomous and accessed through a formal contract, which provides loose coupling and abstraction.
- **Discoverability**. Services can expose descriptions that allow other applications and services to locate them and automatically determine the interface.

Examples
Common examples of service-oriented application include:
- Sharing of medical data (Harvard Medical School)
- Reservation system (Starwood Hotels and Resorts)
- Workflow system (State Children's Health Insurance Program)

Additional Resources
For more information about architectural styles, see the following articles:
- *Fear Those Tiers* at http://msdn.microsoft.com/en-us/library/cc168629.aspx.
- *Layered Versus Client-Server* at http://msdn.microsoft.com/en-us/library/bb421529.aspx.
- *Services Fabric: Fine Fabrics for New-Era Systems* at http://msdn.microsoft.com/en-us/library/cc168621.aspx.
- *Message Bus at http://msdn.microsoft.com/en-us/library/ms978583.aspx*

Chapter 7: Quality Attributes

Objectives

- Learn the key quality attributes, and how they apply to applications.
- Learn the key issues, decisions, and techniques associated with each quality attribute.

Overview

Quality attributes are the cross-cutting concerns that affect run-time performance, system design, and user experience. Quality attributes are important for the overall usability, performance, reliability, and security of software applications. The quality of the application is measured by the extent to which it possesses a desired combination of these quality attributes. When designing applications to meet any of this quality attributes requirements, it is necessary to consider the potential impact on other requirements. You need to analyze the tradeoffs between multiple quality attributes. The importance or priority of each quality attribute differs from system to system; for example, in a line of business (LOB) system, performance, scalability, security, and usability will be more important than interoperability, while in a packaged application, interoperability will be very important.

How to Use This Chapter

This chapter lists and describes the quality attributes that you must consider when you design your application. To get the most out of this chapter, first take into account the objectives and overview above, and then use the table in the section How the Quality Attributes are organized to gain an understanding of how quality attributes map to system and application quality factors. Next, look at the Quality Attribute Frame table, which describes each of the quality attributes. Finally, for each of the quality attributes, understand the lists of the key issues for that attribute, the decisions you must make to addresses these issues, and the key techniques you can use to implement solutions for that quality attribute. Keep in mind that the list of quality attributes in this chapter is not exhaustive, but it provides a good starting point for asking appropriate questions about your architecture.

How the Quality Attributes are Organized

Quality attributes represent areas of concern that have the potential for application-wide impact across layers and tiers. Some of these attributes are related to the overall system design, while others are specific to run-time, design-time, or user-centric issues. Use the following table to gain an understanding of the quality attributes and the scenarios they are most likely to affect.

Type	Quality attributes
System Qualities	• Supportability • Testability

Type	Quality attributes
Run-time Qualities	• Availability • Interoperability • Manageability • Performance • Reliability • Scalability • Security
Design Qualities	• Conceptual Integrity • Flexibility • Maintainability • Reusability
User Qualities	• User Experience / Usability

Quality Attribute Frame

The following table describes the quality attributes covered in this chapter. Use this table to understand what each of the quality attributes means in terms of your application design.

Quality attribute	Description
Availability	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
Conceptual Integrity	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.
Flexibility	Flexibility is the ability of a system to adapt to varying environments and situations, and to cope with changes in business policies and rules. A flexible system is one that is easy to reconfigure or adapt in response to different user and system requirements.
Interoperability	Interoperability is the ability of diverse components of a system or different systems to operate successfully by exchanging information, often by using services. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
Maintainability	Maintainability is the ability of a system to undergo changes to its components, services, features, and interfaces as may be required when adding or changing the functionality, fixing errors, and meeting new business requirements.
Manageability	Manageability defines how easy it is to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
Performance	Performance is an indication of the responsiveness of a system to execute any action within a given time interval. It can be measured in terms of latency or throughput. *Latency* is the time taken to respond to any event. *Throughput* is the number of events that take place within a given amount of time.

Quality attribute	Description
Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified time interval.
Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Scalability	Scalability is the ability of a system to function well when there are changes to the load or demand. Typically, the system will be able to be extended over more powerful or more numerous servers as demand and load increase.
Security	Security defines the ways that a system is protected from disclosure or loss of information, and the possibility of a successful malicious attack. A secure system aims to protect assets and prevent unauthorized modification of information.
Supportability	Supportability defines how easy it is for operators, developers, and users to understand and use the application, and how easy it is to resolve errors when the system fails to work correctly.
Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
Usability	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, and able to provide good access for disabled users and a good overall user experience.

Availability

Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load. Use the techniques listed below to maximize availability for your application.

Key Issues

- A physical tier such as the database server or application server can fail or become unresponsive, causing the entire system to fail.
- Security vulnerabilities can allow Denial of Service (DoS) attacks, which prevent authorized users from accessing the system.
- Inappropriate use of resources can reduce availability. For example, resources acquired too early and held for too long cause resource starvation and an inability to handle additional concurrent user requests.
- Bugs or faults in the application can cause a system-wide failure.
- Frequent updates, such as security patches and user application upgrades, can reduce the availability of the system,

- A network fault can cause the application to be unavailable.

Key Decisions

- How to design failover support related to different tiers in the system.
- How to decide if there is a need for a geographically separate redundant site to failover to in case of natural disasters such as earthquakes or tornados.
- How to design for run-time upgrades.
- How to design for proper exception handling in order to reduce application failures.
- How to handle unreliable network connections.

Key Techniques

- Use Network Load Balancing (NLB) for Web servers in order to distribute the load and prevent requests from being sent to a server that is down.
- Use a Redundant Array of Independent Disks (RAID) to mitigate system failure in the event that a disk fails.
- Deploy the system at geographically separate sites and balance requests across all sites that are available. This is an example of advanced networking design.
- To minimize security vulnerabilities, reduce the attack surface area, identify malicious behavior, use application instrumentation to expose unintended behavior, and implement comprehensive data validation.
- Design clients with occasionally connected capabilities, such as a rich client.

Conceptual Integrity

Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming. A coherent system makes it easy to resolve issues because you will know what is consistent with the overall design. Conversely, a system without conceptual integrity will constantly be affected by changing interfaces, frequently deprecating modules, and lack of consistency in how tasks are performed.

Key Issues

- Mixing different areas of concern together within your design.
- Not using or inconsistent use of a development process.
- Collaboration and communication between different groups involved with the application lifecycle.
- Lack of design and coding standards.
- Existing (legacy) system demands that prevent both refactoring and progression toward a new platform or paradigm.

Key Decisions

- How to identify areas of concern and group them into logical layers.
- How to manage the development process.
- How to facilitate collaboration and communication throughout the application lifecycle.

- How to establish and enforce design and coding standards.
- How to create a migration path away from legacy technologies.
- How to isolate applications from external dependencies.

Key Techniques

- Use published guidelines to help identify areas of concern and group them into logical layers within the design.
- Perform an Application Lifecycle Management (ALM) assessment.
- Establish a development process integrated with tools to facilitate process workflow, communication, and collaboration.
- Establish published guidelines for design and coding standards.
- Incorporate code reviews into your development process to ensure guidelines are being followed.
- Use the Gateway design pattern for integration with legacy systems.
- Provide documentation to explain the overall structure of the application.

Flexibility

Flexibility is the ability of a system to adapt to varying environments and situations, and to cope with changes in business policies and rules. A flexible system is one that can be easily modified in response to different user and system requirements.

Key Issues

- The code base is large, unmanageable, and fragile.
- Refactoring is burdensome due to regression requirements for a large and growing code base.
- The existing code is over-complex.
- The same logic is implemented in many different ways.

Key Decisions

- How to handle dynamic business rules, such as changes related to authorization, data, or process.
- How to handle a dynamic user interface (UI), such as changes related to authorization, data, or process.
- How to respond to changes in data and logic processing.
- How to ensure that components and services have well-defined responsibilities and relationships.

Key Techniques

- Use business components to implement the rules, if only the business rule values tend to change.
- Use an external source, such as a business rules engine, if the business decision rules tend to change.
- Use a business workflow engine if the business process tends to change.

- Design systems as well-defined layers, or areas of concern, that clearly delineate the system's UI, business processes, and data access functionality.
- Design components to be cohesive and loosely coupled to maximize flexibility and facilitate replacement and reusability.

Interoperability

Interoperability is the ability of diverse components of a system or different systems to operate successfully by exchanging information, often by using services. An interoperable system allows you to exchange and reuse information internally as well as externally. Communication protocols, interfaces, and data formats are the key considerations for interoperability. Standardization is also an important aspect to be considered when designing an interoperable system.

Key Issues

- Interaction with external or legacy systems that use different data formats.
- Boundary blurring, which allows artifacts from one layer, tier, or system to defuse into another.

Key Decisions

- How to handle different data formats from external or legacy systems.
- How to enable systems to interoperate while evolving separately or even being replaced.
- How to isolate systems through the use of service interfaces.
- How to isolate systems through the use of mapping layers.

Key Techniques

- Use orchestration with adaptors to connect with external or legacy systems and translate data between systems.
- Use a canonical data model to handle interaction with a large number of different data formats.
- Expose services using interfaces based on XML or standard types in order to support interoperability with other systems.
- Design components to be cohesive and have low coupling in order to maximize flexibility and facilitate replacement and reusability.

Maintainability

Maintainability is the ability of a system to undergo changes to its components, services, features, and interfaces as may be required when adding or changing functionality, fixing bugs, and meeting new business requirements. Measurability can be measured in terms of the time it takes to restore the system to its operational status following a failure or removal from operation for upgrading. Improving system maintainability will increase efficiency and reduce run-time defects.

Key Issues
- Excessive dependencies between components and layers prevent easy replacement, updates, and changes.
- Use of direct communication prevents changes to the physical deployment of components and layers.
- Reliance on custom implementations of features such as authentication and authorization prevents reuse and hampers maintenance.
- Mixing the implementation of cross-cutting concerns with application-specific components makes maintenance harder and reuse difficult.
- Components are not cohesive, which makes them difficult to replace and causes unnecessary dependencies on child components.

Key Decisions
- How to reduce dependencies between components and layers.
- How to implement a pluggable architecture that allows easy upgrades and maintenance, and improved testing capabilities.
- How to separate the functionality for cross-cutting concerns from application-specific code.
- How to choose an appropriate communication model, format, and protocol.
- How to create cohesive components.

Key Techniques
- Design systems as well-defined layers, or areas of concern, that clearly delineate the system's UI, business processes, and data access functionality.
- Design components to be cohesive and have low coupling in order to maximize flexibility and facilitate replacement and reusability.
- Design interfaces that allow the use of plug-in modules or adapters to maximize flexibility and extensibility.
- Provide good architectural documentation to explain the structure of the application.

Manageability
Design your application to be easy to manage, by exposing sufficient and useful instrumentation for use in monitoring systems and for debugging and performance tuning.

Key Issues
- Lack of diagnostic information
- Lack of troubleshooting tools
- Lack of performance and scale metrics
- Lack of tracing ability
- Lack of health monitoring

Key Decisions
- How to enable the system behavior to change based on operational environment requirements, such as infrastructure or deployment changes.

- How to enable the system behavior to change at run time based on system load; for example, by queuing requests and processing them when the system is available.
- How to create a snapshot of the system's state to use for troubleshooting.
- How to monitor aspects of the system's operation and health.
- How to create custom instrumentation to provide detailed operational reports.
- How to discover details of the requests sent to the system.

Key Techniques

- Consider creating a health model that defines the significant state changes that can affect application performance, and use this model to specify management instrumentation requirements.
- Implement instrumentation, such as events and performance counters, that detects state changes, and expose these changes through standard systems such as Event Logs, Trace files, or Windows Management Instrumentation (WMI).
- Capture and report sufficient information about errors and state changes in order to enable accurate monitoring, debugging, and management.
- Consider creating management packs that administrators can use in their monitoring environments to manage the application.
- Consider monitor health of your application or specific functions for debugging.
- Consider logging and auditing information that may be useful for maintenance and debugging, such as request details or module outputs and calls to other systems and services.

Performance

Performance is an indication of the responsiveness of a system to execute specific actions in a given time interval. It can be measured in terms of latency or throughput. Latency is the time taken to respond to any event. Throughput is the number of events that take place in a given amount of time. Factors affecting system performance include the demand for a specific action and the system's response to the demand.

Key Issues

- Increased client response time, reduced throughput, and server resource over-utilization.
- Increased memory consumption, resulting in reduced performance, unable to find data in cache, and increased data store access.
- Increased database server processing may cause reduced throughput.
- Increased network bandwidth consumption may cause delayed response times, and increased load for client and server systems.
- Inefficient queries, or fetching all of the data when only a portion is displayed, may incur unnecessary load on the database server, failure to meet performance objectives, and costs in excess of budget allocations.
- Poor resource management can result in the creation of multiple instances of resources, with the corresponding connection overhead, and can increase the application's response time.

Key Decisions
- How to determine a caching strategy.
- How to design high-performance communication between layers.
- How to choose effective types of transactions, locks, threading, and queuing.
- How to structure the application.
- How to manage resources effectively.

Key Techniques
- Choose the appropriate remote communication mechanism.
- Design coarse-grained interfaces that require the minimum number of calls (preferably just one) to execute a specific task.
- Minimize the amount of data sent over the network.
- Batch work to reduce calls over the network.
- Reduce transitions across boundaries.
- Consider asynchronous communication.

Reliability
Reliability is the ability of a system to continue operating as expected over time. Reliability is measured as the probability that a system will not fail and that it will perform its intended function for a specified time interval. Improving the reliability of a system may lead to a more secure system because it helps to prevent the types of failures that a malicious user may exploit.

Key Issues
- System may crash.
- System becomes unresponsive at times.
- Output is inconsistent.
- System fails because of unavailability of other externalities such as systems, networks, and databases.

Key Decisions
- How to handle unreliable external systems.
- How to detect failures and automatically initiate a failover.
- How to redirect load under extreme circumstances.
- How to take the system offline but still queue pending requests.
- How to handle failed communications.
- How to handle failed transactions.

Key Techniques
- Implement instrumentation, such as events and performance counters, that detects poor performance or failures of requests sent to external systems, and expose information through standard systems such as Event Logs, Trace files, or WMI.
- Log performance and auditing information about calls made to other systems and services.

- Consider implementing configuration settings that change the way the application works, such as using a different service, failing over to another system, or accessing a spare or backup system should the usual one fail.
- Consider implementing code that uses alternative systems when it detects a specific number of failed requests to an existing system.
- Implement store-and-forward or cached message-based communication systems that allow requests to be stored when the target system is unavailable, and replayed when it is online.
- Consider using Windows Message Queuing or Microsoft BizTalk® Server to provide a reliable once-only delivery mechanism for asynchronous requests.

Reusability

Reusability is the probability that a component will be used in other components or scenarios to add new functionalities with little or no change. Reusability minimizes the duplication of components and also the implementation time. Identifying the common attributes between various components is the first step in building small reusable components of a larger system.

Key Issues

- Using different code or components to achieve the same result in different places.
- Using multiple similar methods instead of parameters to implement tasks that vary slightly.
- Using several systems to implement the same feature or function.

Key Decisions

- How to reduce duplication of similar logic in multiple components.
- How to reduce duplication of similar logic in multiple layers or subsystems.
- How to reuse functionality in another system.
- How to share functionality across multiple systems.
- How to share functionality across different subsystems within an application.

Key Techniques

- Examine the application design to identify cross-cutting concerns such as validation, logging, and authentication, and implement these functions as separate components.
- Examine the application design to identify common functionality, and implement this functionality in separate components that you can reuse.
- Consider exposing functionality from components, layers, and subsystems through service interfaces that other layers and systems can use.
- Consider using platform-agnostic data types and structures that can be accessed and understood on different platforms.

Scalability

Scalability is an attribute of a system that displays the ability to function well even with change in demand. Typically, the system should be able to handle increases in size or volume. The aim is to maintain the system's availability, reliability, and performance even when the load increases. There are two methods for improving scalability: scaling vertically, and scaling horizontally. You add more resources such as CPU, memory, disk, etc. to a single system to scale vertically. You add more machines, for serving the application, to scale horizontally.

Key Issues

- Applications cannot handle increasing load.
- Users incur delays in response and longer completion times.
- The system fails.
- The system cannot queue excess work and process it during periods of reduced load.

Key Decisions

- How to design layers and tiers for scalability.
- How to scale up or scale out an application.
- How to scale the database.
- How to scale the UI.
- How to handle spikes in traffic and load.

Key Techniques

- Avoid stateful components and subsystems where possible to reduce server affinity.
- Consider locating layers on the same physical tier to reduce the number of servers required while maximizing load-sharing and failover capabilities.
- Consider implementing configuration settings that change the way the application works, such as using a different service, failing over to another system, or accessing a spare or backup system in case the usual system fails.
- Consider implementing code that uses alternative systems when it detects a specific number of failed requests to an existing system.
- Consider implementing code that uses alternative systems when it detects a predefined service load or a number of pending requests to an existing system.
- Implement store-and-forward or cached message-based communication systems that allow requests to be stored when the target system is unavailable, and replayed when it is online.
- Consider partitioning data across more than one database server to maximize scale-up opportunities and allow flexible location of data subsets.

Security

Security is an attribute of a system that needs to be protected from disclosure or loss of information. Securing a system aims to protect assets and unauthorized modification of information. The factors affecting system security are confidentiality, integrity, and availability. Authentication, encryption, and auditing and logging are the features used for securing systems.

Key Issues

- Spoofing of user identity
- Tampering with data
- Repudiation

- Information disclosure
- Denial of service (DoS)

Key Decisions
- How to address authentication and authorization.
- How to protect against malicious input.
- How to protect sensitive data.
- How to protect against SQL injection.
- How to protect against cross-site scripting.

Key Techniques
- Identify the trust boundaries, and authenticate and authorize users crossing a trust boundary.
- Validate input for length, range, format, and type using constrain, reject, and sanitize principles. Encode output.
- Do not reveal sensitive system or application information.
- Use application instrumentation to expose behavior that can be monitored.
- Partition the site into anonymous, identified, and authenticated users.
- Reduce session timeouts.

Supportability
Supportability is the ability to provide support to a system when it fails to work correctly.

Key Issues
- Lack of diagnostic information
- Lack of troubleshooting tools
- Lack of performance and scale metrics
- Lack of tracing ability
- Lack of health monitoring

Key Decisions
- How to monitor system activity.
- How to monitor system performance.
- How to implement tracing.
- How to provide troubleshooting support.
- How to design auditing and logging.

Key Techniques
- Consider a system monitoring application, such as Microsoft System Center.
- Use performance counters to monitor system performance.
- Enable tracing in Web applications in order to troubleshoot errors.
- Use common components to provide tracing support in code.
- Use Aspect Oriented Programming (AOP) techniques to implement tracing.

Testability

Testability is a measure of how well system or components allow you to create test criteria and execute tests to determine if the criteria are met. Testability allows faults in a system to be isolated in a timely and effective manner.

Key Issues

- Complex applications with many processing permutations are not tested consistently.
- Automated or granular testing cannot be performed because the application has a monolithic design.
- Lack of test planning.
- Poor test coverage—manual as well as automated.
- Input inconsistencies; for the same input, the output is not same.
- Output inconsistencies—output does not fully cover the output domain, even though all known variations of input are provided.

Key Decisions

- How to ensure an early start to testing during the development life cycle.
- How to automate user interaction tests.
- How to handle test automation and detailed reporting for highly complex functionality, rules, or calculations.
- How to separately test each layer or tier.
- How to make it easy to specify and understand system inputs and outputs to facilitate the construction of test cases.
- How to clearly define component and communication interfaces.

Key Techniques

- Use mock objects during testing.
- Construct simple, structured solutions.
- Design systems to be modular to support testing.
- Provide instrumentation or implement probes for testing.
- Provide mechanisms to debug output and ways to specify inputs easily.

User Experience / Usability

The application interfaces must be designed with the user and consumer in mind so that they are intuitive, can be localized and globalized, provide access to disabled users, and provide a good overall user experience.

Key Issues

- Too much interaction (excessive number of “clicks”) is required for a task.
- There is an incorrect flow to the interface.
- Data elements and controls are poorly grouped.
- Feedback to the user is poor, especially for errors and exceptions.
- The application is unresponsive.

Key Decisions

- How to leverage effective interaction patterns.
- How to determine user experience acceptance criteria.
- How to improve responsiveness for the user.
- How to determine the most effective UI technology.
- How to enhance the visual experience.

Key Techniques

- Design the screen and input flows and user interaction patterns to maximize ease of use.
- Incorporate workflows where appropriate to simplify multi-step operations.
- Choose appropriate control types (such as option groups and check boxes) and lay out controls and content using the accepted UI design patterns.
- Implement technologies and techniques that provide maximum user interactivity, such as Asynchronous JavaScript and XML (AJAX) in Web pages and client-side input validation.
- Use asynchronous techniques for background tasks, and tasks such as populating controls or performing long-running tasks.

Additional Resources

For more information on implementing and auditing quality attributes, see the following resources:

- *Implementing System-Quality Attributes* at http://msdn.microsoft.com/en-us/library/bb402962.aspx.
- *Software Architecture in the New Economy* at http://msdn.microsoft.com/en-us/library/cc168642.aspx.
- *Quality-Attribute Auditing: The What, Why, and How* at http://msdn.microsoft.com/en-us/library/bb508961.aspx.

Chapter 8: 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 various 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 (e.g. method calls between components) or message-based communication. There are many advantages to using message-based communication, such as the ability to decouple your components from each other. Decoupling components is not only good for maintainability and but can also give additional flexibility to change your deployment strategy in the future. However, message-based communication raises issues that you must consider, such as performance, reliability, and security.

This chapter presents design guidelines that will help you to choose the appropriate communication approach, understand how to get the best results from your chosen approach, and anticipate security and reliability issues that might 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 the following guidelines when deciding on a communication strategy:

- **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 protocol to package data into a single call that can be serialized across process boundaries.
- **Consider message-based communication when crossing physical boundaries**. Consider using 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 (UI).
- **Consider the serialization capabilities of the data formats passed across boundaries**. If you require interoperability with other systems, consider Extensible Markup Language (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 allows 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.

Consider the following guidelines when using message-based communication:
- Be aware that a connection will not always be present, and that 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.
- Consider using 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 vs. 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 that 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.
- In cases where you must guarantee the order in which operations take place, or where you use operations that depend on the outcome of previous operations, consider a synchronous model.
- For asynchronous, in-process calls use the platform features (such as Begin and End versions of methods and callbacks) to implement asynchronous method calls.
- Consider implementing asynchronous interfaces in the same layer as the caller to obtain maximum responsiveness.
- 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 MSMQ to queue messages for later delivery in case of system or network interruption or failure. MSMQ can perform transacted message delivery and supports reliable once-only delivery.
- Consider using Microsoft BizTalk® Server to interoperate with other systems and platforms at the 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.

Consider the following guidelines when designing for coupling and cohesion:

- For loose coupling, consider using a message-based technology such as ASP.NET Web services (ASMX) or WCF.
- For loose coupling, consider using self-describing data and ubiquitous protocols such as HyperText Transfer Protocol (HTTP) and SOAP.
- To maintain cohesion, ensure that 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.

Consider the following guidelines when deciding how to implement state management:

- 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 stateful 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, consider using 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 the .NET Framework 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 initial 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 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. • You want built-in support for serialization
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 and data formats, such as HTTP and XML..
- Keep in mind that protocol decisions may affect the availability of clients you are targeting. For example, target systems might be protected by firewalls that block some protocols.
- Keep in mind that data format decisions may affect interoperability. For example, target systems might not understand platform-specific types, or might 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 might 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.
- Consider using Data Transfer Objects (DTOs) 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, consider using 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, protect sensitive data from being tampered with, and if necessary, 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) encryption and Internet Protocol Security (IPSec).

Consider the following when deciding whether 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 transport security only.
- If the message passes through one or more servers, use message-based protection as well as transport security. With transport 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 when deciding whether to use message security:

- Always implement message-based security for sensitive messages that pass out of your secure network.
- Always use message-based security for sensitive messages if there are intermediate systems between the client and the service. Intermediate servers will receive the message, handle it, and 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

Windows Communication Foundation (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:

- Consider using WCF to communicate with Web services to achieve interoperability with other platforms that also support SOAP, such as the J2EE-based application servers.
- Consider using WCF to communicate with Web services using messages not based on SOAP for applications with formats such as Really Simple Syndication (RSS).
- Consider using WCF to communicate using SOAP messages and binary encoding for data structures when both the server and the client use WCF.
- Consider using WS-MetadataExchange in SOAP requests to obtain descriptive information about a service, such as its Web Services Description Language (WSDL) definition and policies.
- Consider using WS-Security to implement authentication, data integrity, data privacy, and other security features.
- Consider using WS-Reliable Messaging to implement reliable end-to-end communication, even when one or more Web services intermediaries must be traversed.
- Consider using WS-Coordination to coordinate two-phase commit transactions in the context of Web services conversations.
- Consider using 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, consider using the HTTP protocol.
- When providing interfaces that are accessed from within a private network, consider using the TCP protocol.

- When providing interfaces that are accessed from the same machine, consider using 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 Internet Information Services (IIS) Web server.

ASMX has the following characteristics:

- Can be accessed over the Internet.
- Uses port 80 by default, but this can be easily reconfigured.
- Supports the HTTP protocol only.
- Has no support for Distributed Transaction Coordinator (DTC) transaction flow. You must program long-running transactions using custom implementations.
- Supports IIS authentication.
- Supports Roles stored as Windows groups for authorization.
- Supports IIS and ASP.NET impersonation.
- Supports SSL transport security.
- Supports the endpoint technology implemented in IIS.
- Provides cross-platform interoperability and cross-company computing.

REST vs. SOAP

Representational State Transfer (REST) and SOAP represent two different styles for implementing services. REST is based on HTTP, which means that it works very much like a Web application, so you can take advantage of HTTP support for non-XML MIME types or streaming content from a service request. Service consumers navigating through REST resources interact with URIs the same way a human user might navigate through and interact with Web pages. SOAP is an XML-based messaging protocol that can be used with any communication protocol.

The main difference between these two styles is how the service state machine is maintained. Don’t think of the service state machine as the application or session state; instead, think of it as the different states that an application passes through during its lifetime. With SOAP, movement through different states, such as initialize and update, is encapsulated behind the service endpoint. With REST, the service state machine is explicit and the representation of its state is addressable by URIs, which are HTTP addresses.

While both REST and SOAP can be used with most service implementations, the REST architectural style is better suited for public services or cases where a service can be accessed by unknown consumers. SOAP is much better suited for implementing a Remote Procedure Call (RPC) interface between layers of an application. With SOAP, you are not restricted to HTTP. The WS-* standards, which can be utilized in SOAP, provide a standard and therefore interoperable method of dealing with common messaging issues such as security, transactions, addressing, and reliability. REST can also provide the same type of functionality, but you must create a custom mechanism because few agreed-upon standards currently exist for these areas.

Consider the following guidelines when choosing between REST and SOAP:

- SOAP is a protocol that provides a basic messaging framework upon which abstract layers can be built.
- SOAP is commonly used as an 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 is an architectural style that can be utilize other protocols, such as JavaScript Object Notation (JSON), the Atom publishing protocol, and custom Plain Old XML (POX) formats.
- REST exposes an application and data as a state machine, not just a service endpoint. REST allows standard HTTP calls such as GET and PUT to be used to query and modify the state of the system.
- REST is sometimes described as having a “stateless nature”, meaning that each individual request sent from the client to the server must contain all of the information necessary to understand the request since the server does not store the session state data.

PART III

Layers

In This Part:

- ▶ Layers and Tiers
- ▶ Presentation Layer Guidelines
- ▶ Business Layer Guidelines
- ▶ Data Access Layer Guidelines
- ▶ Service Layer Guidelines

Chapter 9: Layers and Tiers

Objectives

- Learn how to divide your applications into separate physical and logical parts.
- Learn the difference between logical layers and physical tiers.
- Learn about services that you can use to expose logic on layers.
- Learn about the components commonly encountered in layers and tiers.
- Learn about applications that support multiple client types.
- Learn how to choose an appropriate functional layout for your applications.

Overview

This chapter discusses the overall structure for applications, in terms of the logical grouping of components into separate layers or tiers that communicate with each other and with other clients and applications. Layers are concerned with the logical division of components and functionality, and take no account of the physical location of components on different servers or in different locations. Tiers are concerned with the physical distribution of components and functionality on separate servers, computers, networks, and remote locations. Although both layers and tiers use the same set of names (presentation, business, service, and data), remember that only tiers imply a physical separation. It is quite common to locate more than one layer on the same physical machine. You can think of the term “tier” as referring to physical distribution patterns such as two-tier, three-tier, and n-tier.

Layers

Layers are the logical groupings of the software components that make up the application or service. They help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports reusability of components. Each logical layer contains a number of discrete component types grouped into sublayers, with each sublayer performing a specific type of task. By identifying the generic types of components that exist in most solutions, you can construct a meaningful map of an application or service, and then use this map as a blueprint for your design.

Splitting an application into separate layers that have distinct roles and functionalities helps you to maximize maintainability of the code, optimize the way that the application works when deployed in different ways, and provide a clear delineation between locations where certain technology or design decisions must be made.

Presentation, Business, and Data Services

At the highest and most abstract level, the logical architecture view of any system can be considered to be a set of cooperating services grouped into the following layers, as shown in Figure 1.

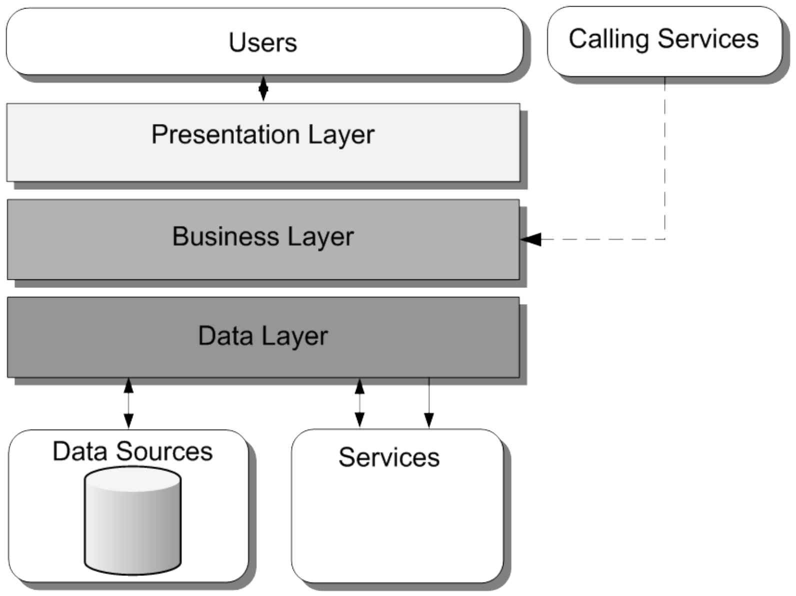

Figure 1 The logical architecture view of a layered system

The sections of the application design shown in Figure 1 can be thought of as three basic sets of services:

- **Presentation services**. These are the user-oriented services responsible for managing user interaction with the system, and generally consist of components located within the presentation layer. They provide a common bridge into the core business logic encapsulated in the business services.
- **Business services**. These services implement the core functionality of the system, and encapsulate the relevant business logic. They generally consist of components located within the business layer, which may expose service interfaces that other callers can use.
- **Data services**. These services provide access to data that is hosted within the boundaries of the system, and data exposed by other back-end systems; perhaps accessed through services. The data layer exposes data to the business layer through generic interfaces designed to be convenient for use by business services.

Components

Each layer of an application will contain a series of components that implement the functionality for that layer. These components should be cohesive and loosely coupled to simplify reuse and maintenance. Figure 2 shows the types of components commonly found in each layer.

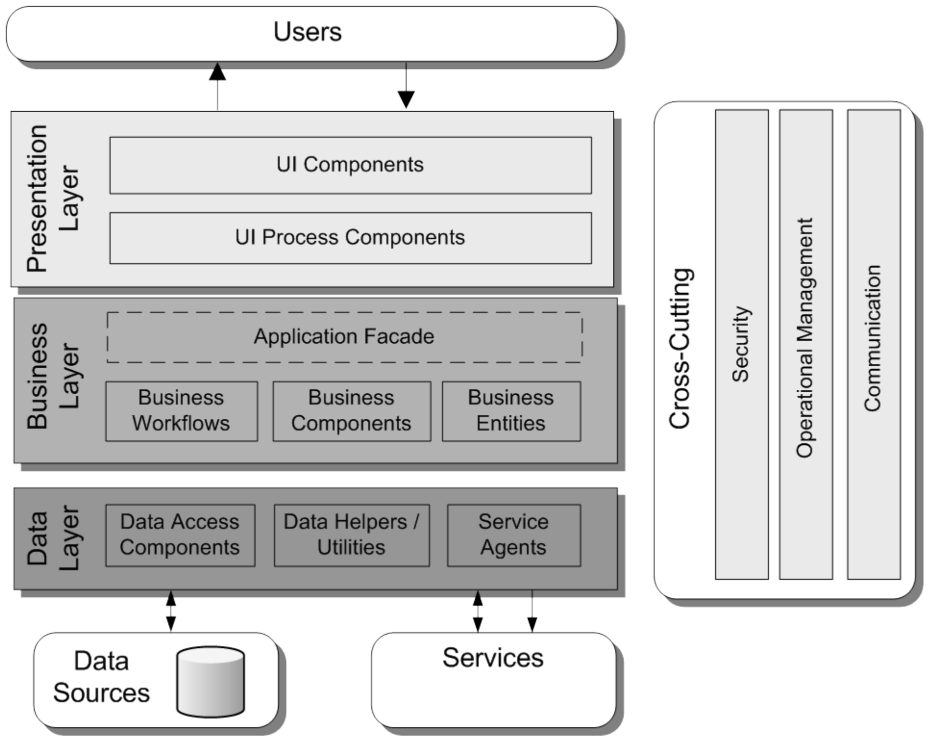

Figure 2 Types of components commonly found in each layer

The components shown in Figure 2 are described in the following sections.

Presentation Layer Components

Presentation layer components implement the functionality required to allow users to interact with the application. The following types of components are commonly found in the presentation layer:

- **User interface (UI) components**. These components provide the mechanism for users to interact with the application. They format data and render it for display, and acquire and validate data entered by users.
- **UI process components**. To help synchronize and orchestrate user interactions, it can be useful to drive the process using separate UI process components. This prevents the process flow and state management logic from being hard-coded into the UI elements themselves, and allows you to reuse the same basic user interaction patterns in other user interfaces.

Business Layer Components

Business layer components implement the core functionality of the system, and encapsulate the relevant business logic. The following types of components are commonly found in the business layer:

- **Application façade.** This is an optional feature that you can use to combine multiple business operations into single messaged-based operations. This feature is useful when you locate the presentation layer components on a separate physical tier from the business layer components, allowing you to optimize use of the communication method that connects them.

- **Business components**. These 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 is likely to require components that implement business rules and perform business tasks.
- **Business workflows**. After the UI components collect the required data from the user and pass it to the business layer, the application can use this data to perform a business process. Many business processes involve multiple steps that must be performed in the correct order, and may interact with each other through an orchestration. Business workflow components define and coordinate long-running, multi-step business processes, and 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 the application uses internally are usually data structures, such as DataSets, DataReaders, or Extensible Markup Language (XML) streams, but they can also be implemented using custom object-oriented classes that represent the real-world entities that your application will handle.

Data Layer Components

Data layer components provide access to data that is hosted within the boundaries of the system, and data exposed by other back-end systems. The following types of components are commonly found in the data layer:

- **Data access components**. These components abstract the logic required to access the underlying data stores. Doing so centralizes data access functionality and makes the application easier to configure and maintain.
- **Data helper and utility components**. Most data access tasks require common logic that can be extracted and implemented in separate reusable helper components. This helps to reduce the complexity of the data access components and centralizes the logic, which simplifies maintenance. Other tasks that are common across data layer components, and not specific to any set of components, may be implemented as separate utility components. Both helper and utility components can often be reused in other applications.
- **Service agents**. When a business component must use functionality provided by an external service, you might need to implement 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.

Cross-Cutting Components

Many tasks carried out by the code of an application are required in more than one layer. Cross-cutting components implement specific types of functionality that can be accessed from components in any layer. The following are common types of cross-cutting components:

- **Components for implementing security**. These may include components that perform authentication, authorization, and validation.

- **Components for implementing operational management tasks**. These may include components that implement exception handling policies, logging, performance counters, configuration, and tracing.
- **Components for implementing communication**. These may include components that communicate with other services and applications.

Services and Layers

From a high-level perspective, a service-based solution can be seen as being composed of multiple services, each communicating with the others by passing messages. Conceptually, the services can be seen as components of the overall solution. However, internally, each service is made up of software components, just like any other application, and these components can be logically grouped into presentation, business, and data services. Other applications can make use of the services without being aware of the way they are implemented. The principles discussed in the previous sections on the layers and components of an application apply equally to service-based solutions.

Services Layer

When an application will act as the provider of services to other applications, as well as implementing features to support clients directly, a common approach is to use a services layer that exposes the functionality of the application, as shown in Figure 3.

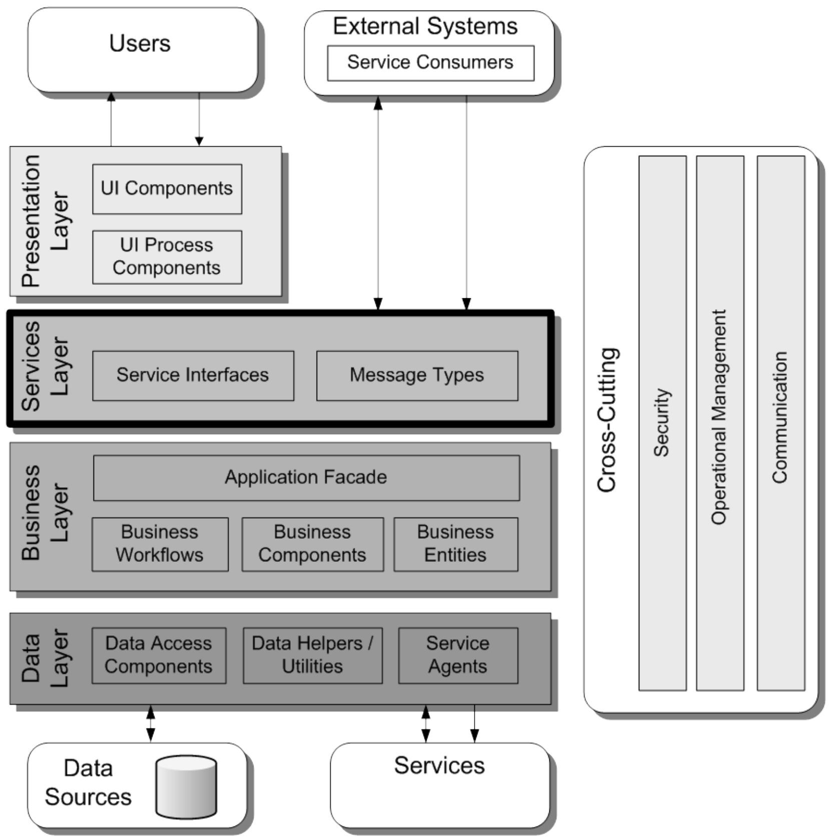

Figure 3 Incorporating a services layer in an application

The following section describes the components usually found in the services layer.

Services Layer Components

Services layer components provide other clients and applications with a way to access business logic in the application, and make use of the functionality of the application by passing messages to and from it over a communication channel. The following types of components are commonly found in the services layer:

- **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.

Multi-Client Application Scenario

Applications often must support different types of clients. In this scenario, the application will usually expose services to external systems, as well as directly supporting local clients, as shown in Figure 4.

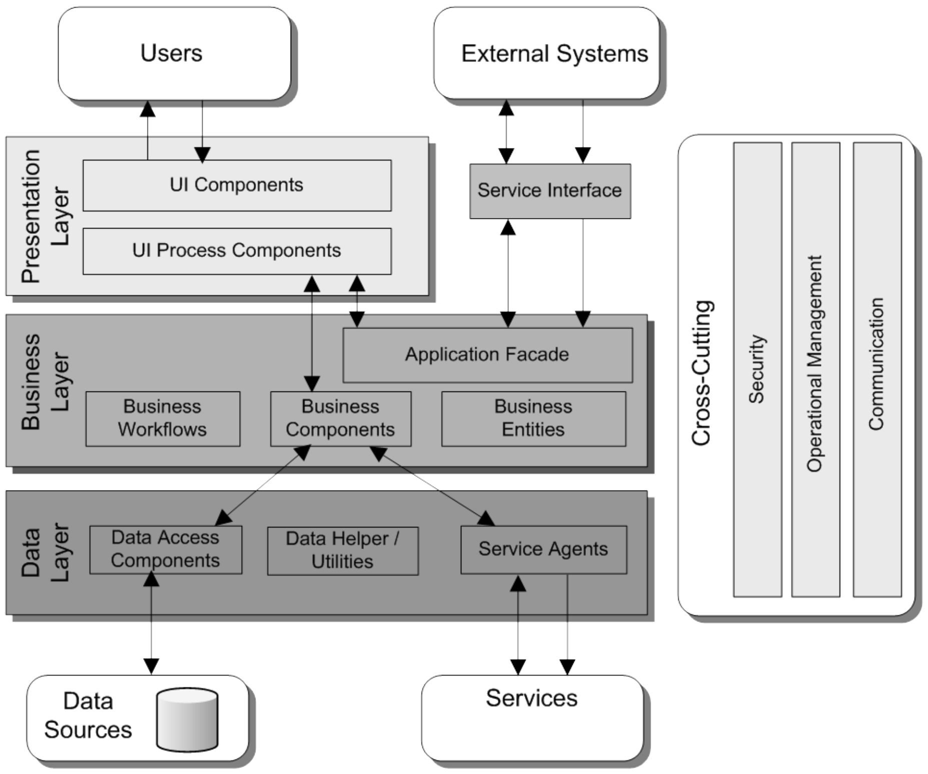

Figure 4 The multi-client application scenario

In this scenario, local and known client types can access the application through the presentation layer, which communicates either directly to the components in the business layer or through an application façade in the business layer if the communication methods require composition of functionality. Meanwhile, external clients and other systems can treat the application as an "application server" and make use of its functionality by communicating with the business layer through service interfaces.

Business Entities Used by Data and Business Services

There are many cases where business entities must be accessible to components and services in both the business layer and the data layer. For example, business entities can be mapped to the data source and accessed by business components. However, you should still separate business logic from data access logic. You can achieve this by moving business entities into a separate assembly that can be shared by both the business services and data services assemblies, as shown in Figure 5. This is similar to using a dependency inversion pattern, where business entities are decoupled from the business and data layer so that both business and data layers are dependent on business entities as a shared contract.

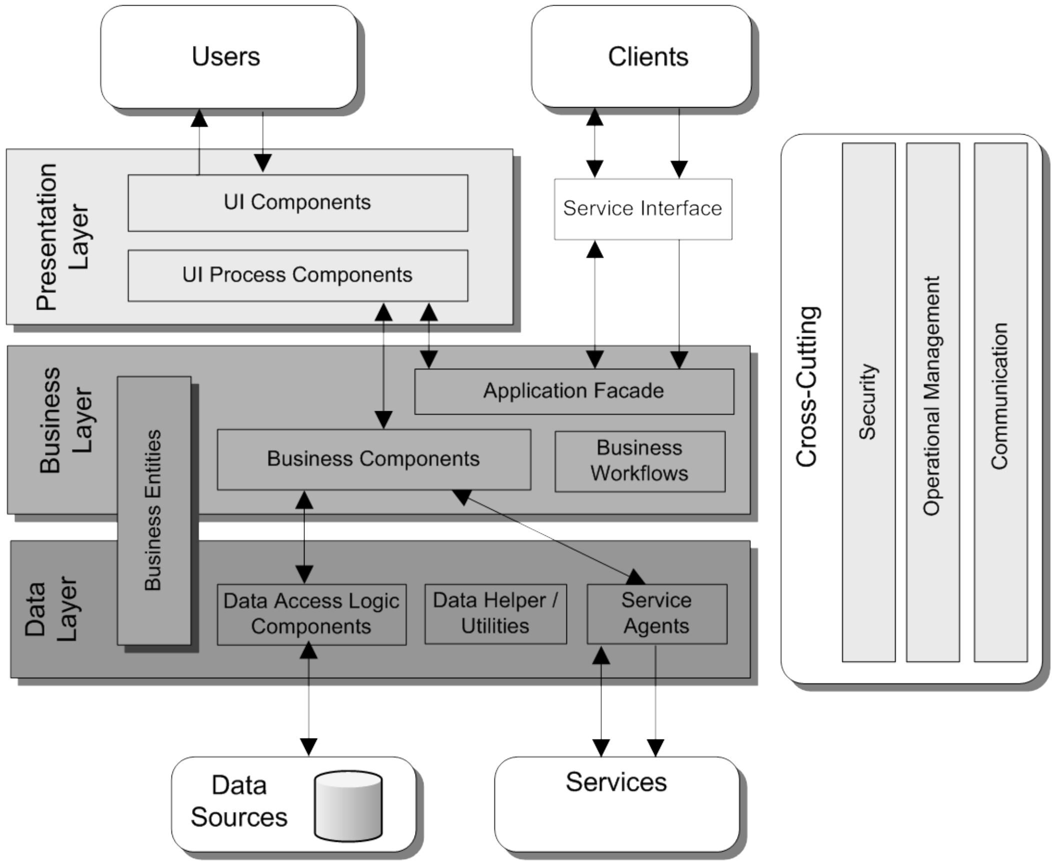

Figure 5 Business entities used by data and business services

Choosing Layers for Your Application

Use a layered approach to improve the maintainability of your application and make it easier to scale out when necessary to improve performance. Keep in mind that a layered approach adds complexity and can impact your initial development time. Be smart about adding layers, and don't add them if you don't need them. Use the following guidelines to help you decide on the layering requirements for your application:

- If your application does not expose a UI, such as a service application, you do not require a presentation layer.
- If your application does not contain business logic, you may not require a business layer.
- If your application does not expose services, you do not require a services layer.
- If your application does not access data, you do not require a data layer.
- Only distribute components where this is necessary. Common reasons for implementing distributed deployment include security policies, physical constraints, shared business logic, and scalability.
- In Web applications, deploy business components that are used synchronously by user interfaces or user process components in the same physical tier as the user interface to maximize performance and ease operational management, unless there are security implications that require a trust boundary between them.
- In rich client applications where the UI processing occurs on the desktop, you may prefer to deploy components that are used synchronously by UIs or user process components in a separate business tier for security reasons, and to ease operational management.
- 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.

Tiers

Tiers represent the physical separation of the presentation, business, services, and data functionality of your design across separate computers and systems. Common tiered design patterns are two-tier, three-tier, and n-tier. The following sections explore each of these scenarios.

Two-Tier

The two-tier pattern represents a basic structure with two main components, a client and a server. In this scenario, the client and server may exist on the same machine, or may be located on two different machines. Figure 6 illustrates a common Web application scenario where the client interacts with a Web server located in the client tier. This tier contains the presentation layer logic and any required business layer logic. The Web application communicates with a separate machine that hosts the database tier, which contains the data layer logic.

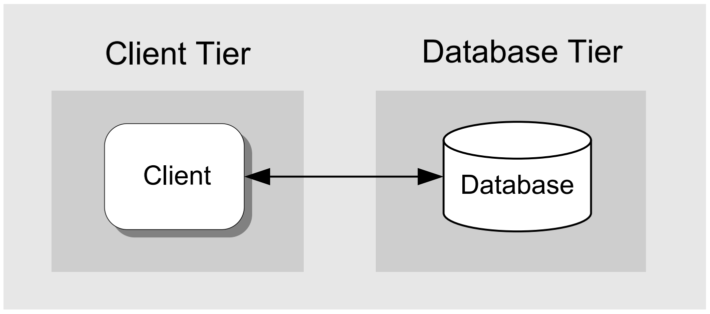

Figure 6 The two-tier deployment pattern

Three-Tier

In a three-tier design, the client interacts with application software deployed on a separate server, and the application server interacts with a database that is also located on a separate server. This is a very common pattern for most Web applications and Web services. Figure 7 illustrates the three-tier deployment pattern.

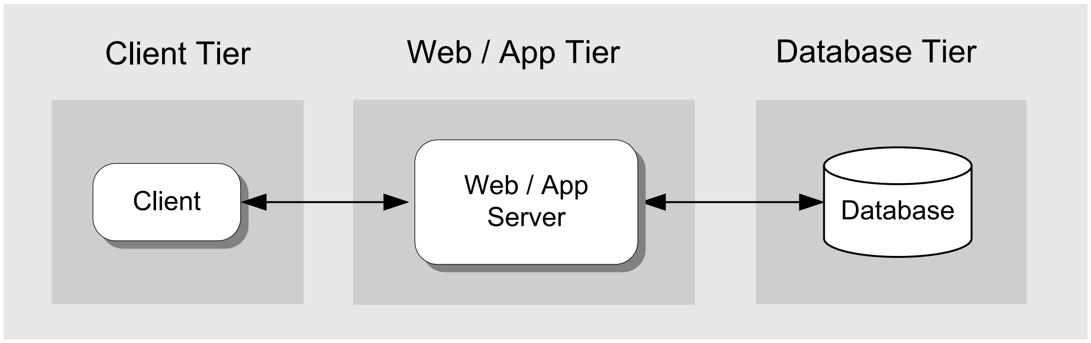

Figure 7 The three-tier deployment pattern

N-Tier

In this scenario, the Web server (which contains the presentation layer logic) is physically separated from the application server that implements the business logic. This usually occurs for security reasons, where the Web server is deployed within a perimeter network and accesses the application server located on a different subnet through a firewall. It is also common to implement a firewall between the client and the Web tier. Figure 8 illustrates the n-tier deployment pattern.

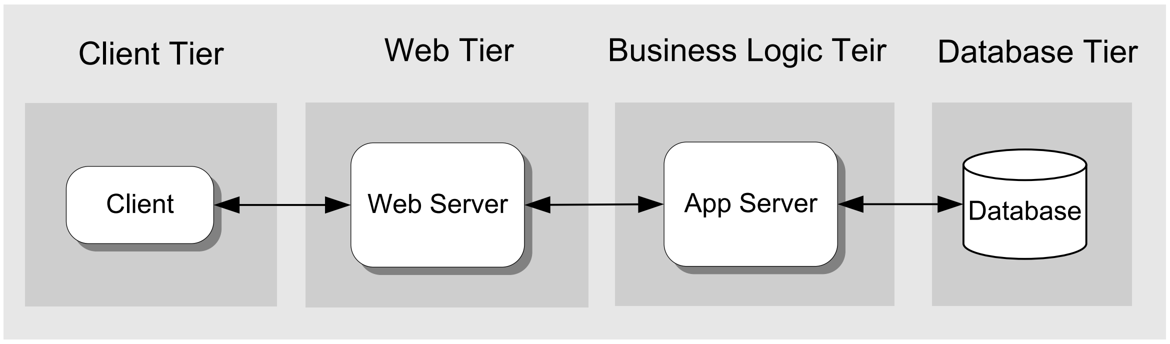

Figure 8 The n-tier deployment pattern

Choosing Tiers for Your Application

Placing your layers on separate physical tiers can help performance by distributing the load across multiple servers. It can also help with security by segregating more sensitive components and layers onto different networks or on the Internet versus an intranet. Keep in mind that adding tiers increases the complexity of your deployment, so don't add more tiers than you need.

In most cases, you should locate all of the application code on the same server, using a single-tier approach. Whenever communications must cross physical boundaries, performance is affected because the data must be serialized. However, in some cases you might need to split functionality across servers, because of security or organizational constraints. To mitigate the effects of serialization, depending on where servers are located, you can usually choose communication protocols that are optimized for performance.

Consider the client/server or two-tier pattern if:

- You are developing a client that must access an application server.
- You are developing a stand-alone client that accesses an external database.

Consider the three-tier pattern if:

- You are developing an intranet-based application, where all servers are located within a private network.
- You are developing an Internet-based application, and security requirements do not restrict implementation of business logic on the public-facing Web or application server.

Consider the N-tier pattern if:

- Security requirements dictate that business logic cannot be deployed to the perimeter network.
- You have application code that makes heavy use of resources on the server, and you want to offload that functionality to another server.

Chapter 10: Presentation Layer Guidelines

Objectives

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

Overview

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

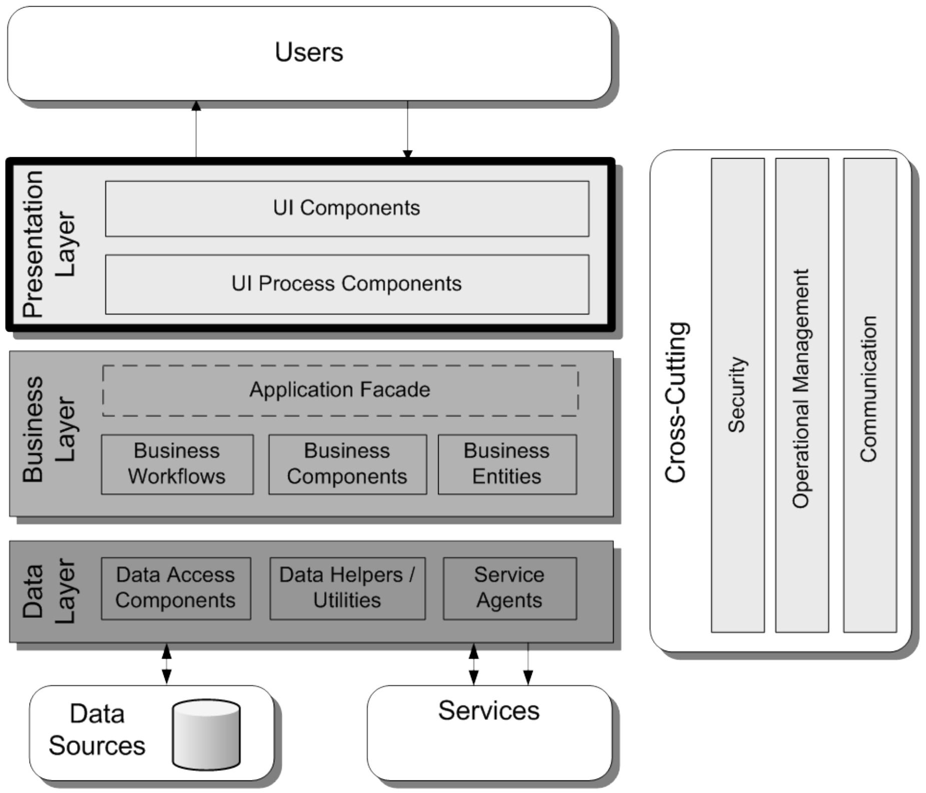

Figure 1 A typical application showing the presentation layer and the components it may contain

Presentation Layer Components

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

Approach

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

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

Design Considerations

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

- **Choose the appropriate UI technology.** Determine if you will implement a rich (smart) client, a Web client, or a rich Internet application (RIA). Base your decision on application requirements, and on organizational and infrastructure constraints.
- **Use the relevant patterns.** Review the presentation layer patterns for proven solutions to common presentation problems.
- **Design for separation of concerns.** Use dedicated UI components that focus on rendering and display. Use dedicated 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 UI design. Review established UI guidelines based on the client type and technologies that you have chosen.
- **Adhere to user-driven design principles.** Before designing your presentation layer, understand your customer. Use surveys, usability studies, and interviews to determine the best presentation design to meet your customer's requirements.

Presentation Layer Frame

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

Category	Common issues
Caching	• Caching volatile data. • Caching unencrypted sensitive data. • Incorrect choice of caching store. • 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 control application flow. • Catching exceptions you do not handle. • Using custom exceptions when not necessary.
Input	• Failing to design for intuitive use, or implementing overly complex interfaces. • Failing to design for accessibility. • Failing to design for different screen sizes and resolutions. • Failing to design for different device and input types, such as mobile devices, touch-screen, and pen and ink–enabled devices.

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

Caching

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

Consider the following guidelines when designing your caching strategy:

- 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 need to be synchronized; instead, consider using a transactional resource manager such as Microsoft 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 run time. Composition patterns support the creation of views and the presentation layout at run time. 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.

Consider the following guidelines when designing your composition strategy:

- Avoid using dynamic layouts. They can be difficult to load and maintain.
- Be careful with dependencies between components. For example, 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 in order 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.

Consider the following guidelines when designing your exception management strategy:

- Use user-friendly error messages to notify users of errors in the application.
- Avoid exposing sensitive data in error pages, error messages, log files, and audit files.
- Design a global exception handler that displays a global error page or an error message for all unhandled exceptions.
- Differentiate between system exceptions and business errors. In the case of business errors, display a user-friendly error message and allow the user to retry the operation. In the case

of system exceptions, check to see if the exception was caused by issues such as system or database failure, display a user-friendly error message, and log the error message, which will help in troubleshooting.
- Avoid using exceptions to control application logic.

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

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

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

Consider the following guidelines when designing your layout strategy:
- Use templates to provide a common look and feel to all of the UI screens.
- Use a common look and feel for all elements of your UI to maximize accessibility and ease of use.
- Consider device-dependent input, such as touch screens, ink, or speech, in your layout. For example, with touch-screen input you will typically use larger buttons with more spacing between them than you would with mouse or keyboard inputs.
- When building a Web application, consider using Cascading Style Sheets (CSS) for layout. This will improve rendering performance and maintainability.
- Use design patterns, such as Model-View-Presenter (MVP), to separate the layout design from interface processing.

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

Consider the following guidelines when designing your navigation strategy:
- Use well-known design patterns to decouple the UI from the navigation logic where this logic is complex.
- Design toolbars 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 datasets are sufficiently large and complex to require separate storage from the UI controls.

Consider the following guidelines when designing presentation entities:
- Determine if you require presentation entities. Typically, you may require presentation 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.

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

User Experience

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

Consider the following guidelines when designing for user experience:

- When developing a rich Internet application (RIA), avoid synchronous processing where possible.
- When developing a Web application, consider using Asynchronous JavaScript and XML (AJAX) to improve responsiveness and to reduce post backs and page reloads.
- Do not design overloaded or overly complex interfaces. Provide a clear path through the application for each key user scenario.
- Design to support user personalization, localization, and accessibility.
- Design for user empowerment. Allow the user to control how he or she interacts with the application, and how it displays data to them.

UI Components

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

Consider the following guidelines when designing UI components:

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

UI Process Components

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

Consider the following guidelines when designing UI processing components:

- Do not create UI process components unless you need them.

- If your UI requires complex processing or needs to talk to other layers, use UI process components to decouple this processing from the UI.
- Consider dividing UI processing into three distinct roles: Model, View, and Controller/Presenter, by using the MVC or MVP pattern.
- Avoid business rules, with the exception of input and data validation, in UI processing components.
- Consider using abstraction patterns, such as dependency inversion, when UI processing behavior needs to change based on the run-time environment.
- Where the UI requires complex workflow support, create separate workflow components that use a workflow system such as Windows Workflow or a custom mechanism.

Validation

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

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

- Validate all input data on the client side where possible to improve interactivity and reduce errors caused by invalid data.
- Do not rely on client-side validation only. Always use server-side validation 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, when working with .NET Framework.
- In Web applications, 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 • Command Pattern
Presentation Entities	• Entity Translator
User Experience	• Asynchronous Callback • Chain of Responsibility

Category	Relevant patterns
UI Processing Components	• Model-View-Controller (MVC) • Passive View • Presentation Model • 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.
- **Command Pattern.** Encapsulate request processing in a separate command object with a common execution interface.
- **Entity Translator.** An object that transforms message data types into business types for requests, and reverses the transformation for responses.
- **Exception Shielding.** Prevent a service from exposing information about its internal implementation when an exception occurs.
- **Front Controller**. Consolidate request handling by channeling all requests through a single handler object, which can be modified at run time with decorators.
- **Model-View-Controller**. Separate the UI code into three separate units: Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Page 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.

Technology Considerations

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

Mobile Applications

Consider the following guidelines when designing a mobile application:

- If you want to build full-featured connected, occasionally connected, and disconnected executable applications that run on a wide range of Microsoft Windows®–based devices, consider using the Microsoft Windows Compact Framework.
- If you want to build connected applications that require Wireless Application Protocol (WAP), compact HTML (cHTML), or similar rendering formats, consider using ASP.NET Mobile Forms and Mobile Controls.
- If you want to build applications that support rich media and interactivity, consider using Microsoft Silverlight® for Mobile.

Rich Client Applications

Consider the following guidelines when designing a rich client application:

- If you want to build applications with good performance and interactivity, and have design support in Microsoft Visual Studio®, consider using Windows Forms.
- If you want to build applications that fully support rich media and graphics, consider using Windows Presentation Foundation (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 a Microsoft 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 the Presentation Model (Model-View-ViewModel) pattern.
- 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.

Rich Internet Applications (RIA)

Consider the following guidelines when designing an RIA:

- If you want to build browser-based, connected applications that have broad cross-platform reach, are highly graphical, and support rich media and presentation features, consider using Silverlight.
- If you decide to use Silverlight, consider using the Presentation Model (Model-View-ViewModel) pattern.

Web Applications

Consider the following guidelines when designing a Web application:

- If you want to build applications that are accessed through a Web browser or specialist user agent, consider using ASP.NET.
- If you want to build applications that provide increased interactivity and background processing, with fewer page reloads, consider using ASP.NET with AJAX.
- If you want to build applications that include islands of rich media content and interactivity, consider using ASP.NET with Silverlight controls.
- If you are using ASP.NET and want to implement a control-centric model with separate controllers and improved testability, consider using the ASP.NET MVC Framework.
- If you are using ASP.NET, consider using master pages to simplify development and implement a consistent UI across all pages.

patterns & practices Solution Assets

- Web Client Software Factory at http://msdn.microsoft.com/en-us/library/bb264518.aspx
- Smart Client Software Factory at http://msdn.microsoft.com/en-us/library/aa480482.aspx
- Composite Application Guidance for WPF at http://msdn.microsoft.com/en-us/library/cc707819.aspx
- Smart Client - Composite UI Application Block at http://msdn.microsoft.com/en-us/library/aa480450.aspx

Additional Resources

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

Chapter 11: Business Layer Guidelines

Objectives

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

Overview

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

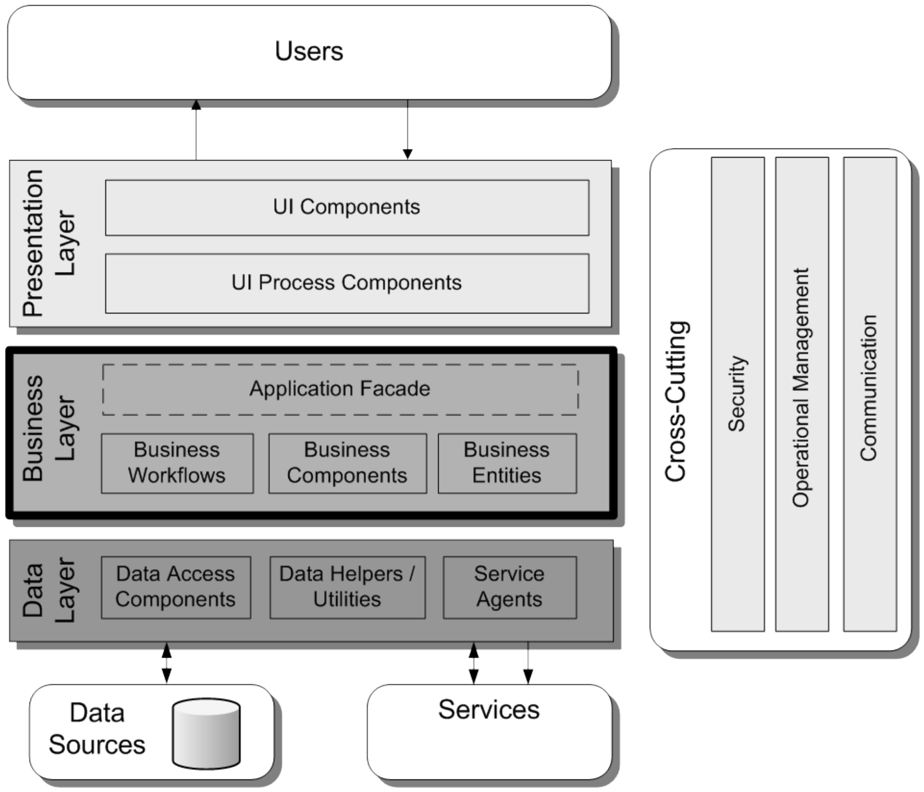

Figure 1 A typical application showing the business layer and the components it may contain

Key Business Components

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

- **Application façade** (optional). An application façade combines multiple business operations into a single message-based operation. You might access the application façade from the presentation layer by using different communication technologies.
- **Business components**. Within the business layer there are different components that provide business services, such as processing business rules and interacting with data access components. For example, you might have a business component that implements the transaction script pattern, which allows you to execute multiple operations within a single component used to manage the transaction. Another business component might be used to process requests and apply business rules.
- **Business entities**. Business components used to pass data between other components are considered business entities. The data can represent real-world business entities, such as products and orders, or database entities, such as tables and views. The business entities that an application uses internally can be implemented using custom objects that represent real-world or database entities your application has to work with. Alternatively, business entities can be implemented using data structures such as DataSets and Extensible Markup Language (XML) documents.
- **Business workflows**. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. Business workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.

Approach

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

1. **Create an overall design for your business layer:**
    - Identify the consumers of your business layer.
    - Determine how you will expose your business layer.
    - Determine the security requirements for your business layer.
    - Determine the validation requirements and strategy for your business layer.
    - Determine the caching strategy for your business layer.
    - Determine the exception-management strategy for your business layer.
2. **Design your business components**:
    - Identify business components your application will use.
    - Make key decisions about location, coupling, and interactions for business components.
    - Choose appropriate transaction support.
    - Identify how your business rules are handled.

- Identify patterns that fit the requirements.

3. **Design your business entity components**:
   - Identify common data formats for the business entities.
   - Choose the data format.
   - Optionally, choose a design for your custom objects.
   - Optionally, determine how you will serialize the components.
4. **Design your workflow components:**
   - Identify workflow style using scenarios.
   - Choose an authoring mode.
   - Determine how rules will be handled.
   - Choose a workflow solution.
   - Design business components to support workflow.

Design Considerations

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

Consider the following guidelines when designing the business layer:

- **Decide if you need a separate business layer**. It is always a good idea to use a separate business layer where possible to improve the maintainability of your application.
- **Identify the responsibilities of your business layer**. Use a business layer for processing complex business rules, transforming data, applying policies, and for validation.
- **Do not mix different types of components in your business layer**. Use a business layer to decouple business logic from presentation and data access code, and to simplify the testing of business logic.
- **Reuse common business logic**. Use a business layer to centralize common business logic functions and promote reuse.
- **Identify the consumers of your business layer**. This will help to determine how you expose your 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 the business layer. • Mixing data access logic within business logic in business components.
Exception Management	• Revealing sensitive information to the end user. • Using exceptions to control application flow. • Not logging sufficient detail from exceptions. • Failing to appropriately notify users with useful error messages.

Category	Common issues
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. • Failure to validate for length, range, format and type. • Not reusing the validation logic.
Workflows	• Not considering application management requirements. • Choosing an incorrect workflow pattern. • Not considering how to handle all exception states. • Choosing an incorrect workflow technology.

Authentication

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

Consider the following guidelines when designing an authentication strategy:

- 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.
- If the presentation and business layers are deployed on 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 is absolutely necessary, as many environments do not 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 web service being called 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. Failure to do so can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges.

Consider the following guidelines when designing an authorization strategy:

- Protect resources by applying authorization to callers based on their identity, account groups, or roles.
- Consider using role-based authorization for business decisions.
- Consider using resource-based authorization for system auditing.
- Consider using 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 because 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.

Consider the following guidelines when designing business components:

- Avoid mixing data access logic and business logic within your business components.
- Design components to be highly cohesive. In other words, you should not overload business components by adding unrelated or mixed functionality.
- If you want to keep business rules separate from business data, consider using business process components to implement your business rules.
- If your application has volatile business rules, store them in a rules engine.
- If the business process involves multiple steps and long-running transactions, consider using workflow components.

Business Entities

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

Consider the following guidelines when designing business entities:

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

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

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

Consider the following guidelines when designing for coupling and cohesion:
- Avoid circular dependencies. The business layer should know only about the layer below (the data access layer), and not the layer above (the presentation layer or external applications that access the business layer directly).
- Use abstraction to implement a loosely coupled interface. This can be achieved with interface components, common interface definitions, or shared abstraction where concrete

components depend on abstractions and not on other concrete components (the principle of Dependency Inversion).
- Design for tight coupling within the business layer unless dynamic behavior requires loose coupling.
- Design for high cohesion. Components should contain only functionality specifically related to that component.
- Avoid mixing data access logic with business logic in your business components.

Concurrency and Transactions

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

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

Data Access

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

Consider the following guidelines when designing a data-access strategy:
- Avoid mixing data-access code and business logic within your business components.
- Avoid directly accessing the database from your business layer.
- Consider using a separate data access layer for access to the database.

Exception Management

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

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

- Do not use exceptions to control business logic.
- Only catch 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.

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

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

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 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.

Consider the following guidelines when designing a service interface:

- Design your service interfaces in such a way that changes to the business logic do not affect the interface.
- Do not implement business rules in a service interface 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. Failure to do so can leave your application vulnerable to cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attacks. There is no comprehensive definition of what constitutes a valid input or malicious input. In addition, how your application uses input influences the risk of the exploit.

Consider the following guidelines when designing a validation strategy:

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

Workflows

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

Consider the following guidelines when designing a workflow strategy:

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

- If the process steps can be executed asynchronously in any order, consider using the event pattern.

Deployment Considerations

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

Consider the following guidelines when deploying a business layer:

- Deploy the business layer to the same physical tier as the presentation or service layer in order to maximize application performance.
- If you must support a remote business layer, consider using the TCP protocol to improve application performance.
- Consider using Internet Protocol Security (IPSec) to protect data passed between physical tiers for all business layers for all applications.
- Consider using Secure Sockets Layer (SSL) encryption 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 of the select, insert, update, and delete queries.
- **Table Module**. A single component that handles the business logic for all rows in a database table or view.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

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

- If you require workflows that automatically support secure, reliable, transacted data exchange, a broad choice of transport and encoding options, and provide built-in persistence and activity tracking, consider using Windows Workflow (WF).
- If you require workflows that implement complex orchestrations and support reliable store and forward messaging capabilities, consider using Microsoft BizTalk® Server.
- If you must interact with non-Microsoft systems, perform electronic data interchange (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 Microsoft Office SharePoint® site and does not require access to information in other sites, consider using Microsoft Office SharePoint Server (MOSS). Note that 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 *Concurrency Control* at http://msdn.microsoft.com/en-us/library/ms978457.aspx.
- For more information, see *Integration Patterns* at http://msdn.microsoft.com/en-us/library/ms978729.aspx.

Chapter 12: Data Access Layer Guidelines

Objectives

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

Overview

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

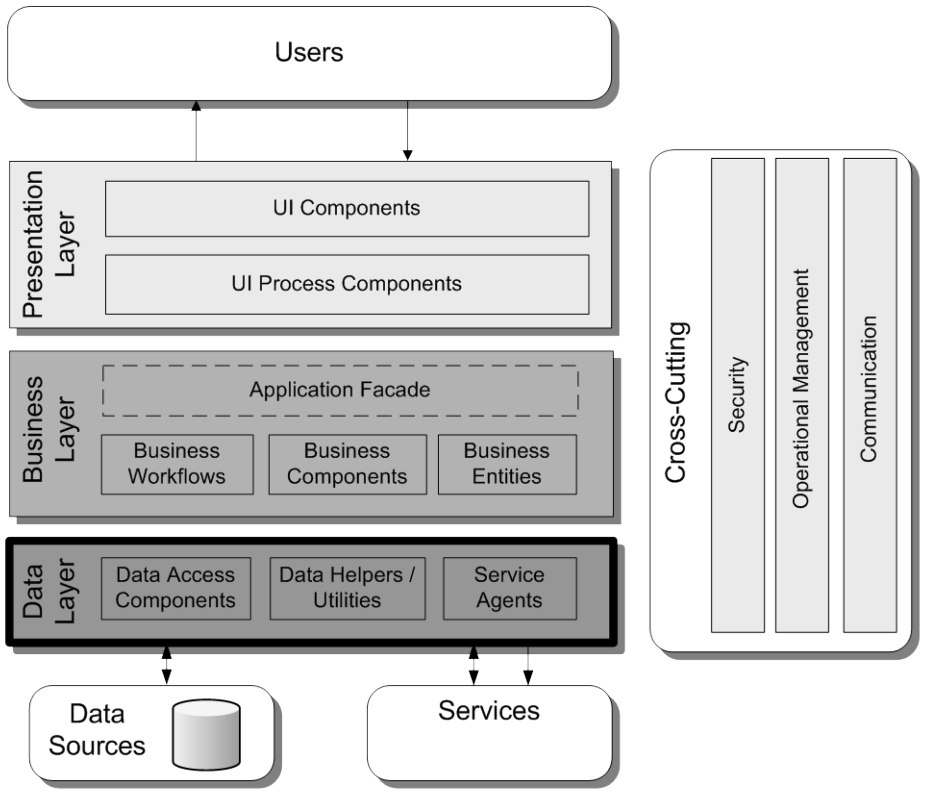

Figure 1 A typical application showing the data layer and the components it may contain

Data Layer Components

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

Approach

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

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

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

Design Guidelines

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

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

- **Determine how you will handle data exceptions**. The data access layer should catch and (at least initially) handle all exceptions associated with data sources and CRUD (Create, Read, Update, and Delete) operations. Exceptions concerning the data itself, and data source access and timeout errors, should be handled in this layer and passed to other layers only if the failures affect application responsiveness or functionality.
- **Consider security risks**. The data access layer should protect against attacks that try to steal or corrupt data, and protect the mechanisms used to gain access to the data source. It should also use the "least privilege" design approach to restrict privileges to only those needed to perform the operations required by the application. If the data source itself has the ability to limit privileges, security should be considered and implemented in the data access layer as well as in the source.
- **Reduce round trips**. Consider batching commands into a single database operation.
- **Consider performance 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 the database. • Searching and manipulating BLOB data.
Batching	• Failing to use batching to reduce database round trips. • Holding onto locks for excessive periods when batching.
Connections	• 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	• Using an incorrect strategy to pass parameters to stored procedures. • Formatting data for display to users 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 block access to data.
Validation	• Failing to validate and constrain data fields. • Not handling NULL values. • Not filtering for invalid characters.
XML	• Not considering how to handle extremely large XML data sets. • Not choosing the appropriate technology for XML to relational database interaction. • 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. A BLOB may have structure within it, but that structure is not apparent to the database that stores it or the data layer that reads and writes it. Databases can store the BLOB data or can store pointers to them within the database. The BLOB data is usually stored in a file system if not stored directly in the database. BLOBs are typically used to store image data, but can also be used to store binary representations of objects.

Consider the following guidelines when designing for BLOBs:

- Store BLOB data in a database only when it is not practical to store it on the disk.
- Consider using BLOBs to simplify synchronization of large binary objects between servers.
- Consider whether you need to search the BLOB data. If so, create and populate other searchable database fields instead of parsing the BLOB data.
- When retrieving the BLOB, cast it to the appropriate type for manipulation within your business or presentation layer.
- Do not consider storing the 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 can improve performance because the database caches and can reuse a query execution plan for a similar query.

Consider the following guidelines when designing batching:

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

Connections

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

Consider the following guidelines when designing for data layer connections:

- In general, open connections as late as possible and close them as early as possible.
- To maximize the effectiveness of connection pooling, consider using a trusted subsystem 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 in order to properly interpret the raw bytes stored in the database and transferred by the data layer. Choosing the appropriate data format provides interoperability with other applications, and facilitates serialized communications across different processes and physical machines. Data format and serialization are also important in order to allow the storage and retrieval of application state by the business layer.

Consider the following guidelines when designing your data format:

- In most cases, you should use custom data or business entities for improved application maintainability. This will require additional code to map the entities to database operations. However, new object/relational mapping (O/RM) solutions are available to reduce the amount of custom code required.
- Consider using XML for interoperability with other systems and platforms or when working with data structures that can change over time.
- Consider using DataSets for disconnected scenarios in simple CRUD-based applications.
- Understand the serialization and interoperability requirements of your application.

Exception Management

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

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

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

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 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.

Consider the following guidelines when designing for object relational mapping:

- 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.
- Consider using objects to build queries. For example, implement the Query Object pattern or use the object support provided by ADO.NET.
- When building dynamic SQL, avoid mixing business-processing logic with logic used to generate the SQL statement. Doing so can lead to code that is very difficult to maintain and debug.

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.

Consider the following guidelines when designing stored procedures:
- 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 procedure.
- 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.

Consider the following guidelines when choosing between stored procedures and dynamic SQL:

- 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 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, 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 that are 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. As a result, you should implement an abstraction in the data access layer to decouple business components from the generation of database queries. Several patterns, such as Query Object and Repository, can be used to provide this abstraction.

Transactions

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

Consider the following guidelines when designing transactions:

- Enable transactions only when you need them. For example, you should not use a transaction for an individual SQL statement because Microsoft SQL Server® 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 and from third-party components, as well as from the database or data store. Understand your trust boundaries so that you can validate any data that crosses these boundaries.

Consider the following guidelines when designing a validation strategy:

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

XML

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

Consider the following guidelines when designing for the use of XML:

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

Manageability Considerations

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

Consider the following guidelines when designing for manageability:

- Consider using common interface types or a shared abstraction (Dependency Inversion) to provide an interface to the data access layer.
- Consider the use of custom entities, or decide if other data representations will better meet your requirements. Coding custom entities can increase development costs; however, they also provide improved performance through binary serialization and a smaller data footprint.
- Implement business entities by deriving them from a base class that provides basic functionality and encapsulates common tasks. However, be careful not to overload the base class with unrelated operations, which would reduce the cohesiveness of entities derived from the base class, and cause maintainability and performance issues.
- Design business entities to rely on data access logic components for database interaction. Centralize implementation of all data access policies and related business logic. For example, if your business entities access SQL Server databases directly, all applications deployed to clients that use the business entities will require SQL connectivity and logon permissions.
- Consider using 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 large trees of stored procedures that call each other.

Performance Considerations

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

Consider the following guidelines when designing for performance:

- 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.
- Consider using optimistic concurrency with non-volatile data to mitigate the cost of locking data in the database. This avoids the overhead of locking database rows, including the connection that must be kept open during a lock.
- If using a DataReader, use ordinal lookups for faster performance.

Security Considerations

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

Consider the following guidelines when designing for security:

- When using Microsoft SQL Server, consider using Windows authentication with a trusted subsystem.
- 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.

Consider the following guidelines when deploying the data access layer:

- 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 • Data Mapper • Data Transfer Object • Domain Model • Query 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 Mapper.** Implement a mapping layer between objects and the database structure that is used to move data from one structure to another while keeping them independent.
- **Data Transfer Object**. An object that stores the data transported between processes, reducing the number of method calls required.
- **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.
- **Query Object**. An object that represents a database query.
- **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 of the select, insert, update, and delete queries.
- **Table Module**. A single component that handles the business logic for all rows in a database table or view.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology and 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 patterns & practices 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=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 13: 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 when designing the service layer.
- Learn the key guidelines for designing the service layer.
- Learn the key patterns and technology considerations for designing the service layer.

Overview

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 into the overall design of your application.

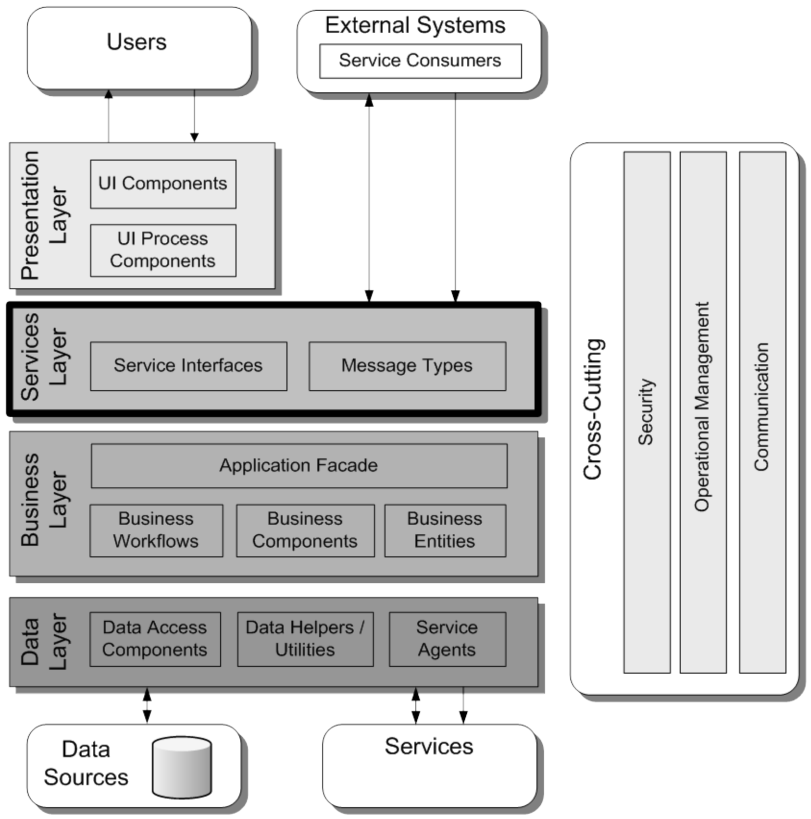

Figure 1 An overall view of a typical application showing the service layer

Service Layer Components

The service layer is made up of the following components:

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

Approach

The approach used to design a service layer starts by defining the service interface, which consists of the contracts that you plan to expose from your service. Once the service interface has been defined, the next step is to design the service implementation; which is used to translate data contracts into business entities and to interact with the business layer.

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

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

Design Considerations

There are many factors that you should consider when designing the service layer. Many of these design considerations relate to proven practices concerned with layered architectures. However, with a service, you must take into account message-related factors. The main thing to consider is that a service uses message-based interaction, which is inherently slower than object-based interaction. In addition, messages passed between a service and a consumer can be routed, modified, or lost, which requires a design that will account for the non-deterministic behavior of messaging.

Consider the following guidelines when designing the service layer:

- **Design services to be application-scoped and not component-scoped**. Service operations should be coarse-grained and focused on application operations. For example, with demographics data, you should provide an operation that returns all of the data in one call. You should not use multiple operations to return subsets of the data with multiple calls.
- **Design entities for extensibility**. In other words, data contracts should be designed so that you can extend them without affecting consumers of the service.

- **Compose entities from standard elements**. When possible, use standard elements to compose the complex types used by your service.
- **Use a layered approach to designing services**. Separate the business rules and data access functions into distinct components where appropriate.
- **Avoid tight coupling across layers**. Use abstraction to provide an interface into the business layer. This abstraction can be implemented by using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation and business layers.
- **Design 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 the service contract**. In other words, you should not implement functionality that is not reflected by the service contract. In addition, the implementation of a service should never be exposed to external consumers.
- **Design to assume the possibility of invalid requests**. You should never assume that all messages received by the service will be valid.
- **Separate functional business concerns from infrastructure operational concerns**. Cross-cutting logic should never be combined with application logic. Doing so can lead to implementations that are difficult to extend and maintain.
- **Ensure that the service can detect and manage repeated messages (idempotency)**. When designing the service, implement well-known patterns to ensure that duplicate messages are not processed.
- **Ensure that the service can manage messages arriving out of order (commutativity)**. If there is a possibility that messages will arrive out of order, implement a design that will store messages and then process them in the correct order.
- **Consider versioning of contracts**. A new version for service contracts mean new operations exposed by the service, whereas for data contracts it means the addition of new schema type definitions.

Service Layer Frame

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

Area	Key issues
Authentication and Authorization	• Lack of authentication across trust boundaries. • Lack of authorization across trust boundaries. • Granular or improper authorization.
Communication	• Incorrect choice of transport protocol. • Use of a chatty service communication interface. • Failing to protect sensitive data.
Exception Management	• Not catching exceptions that can be handled. • Not logging exceptions. • Not dealing with message integrity when an exception occurs.

Area	Key issues
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	• Implementing state within the service. • Overusing POST statements. • Putting actions in the URI. • Using hypertext to manage state.
SOAP	• Not choosing the appropriate security model. • Not planning for fault conditions. • Using complex types in the message schema.

Authentication

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

Consider the following guidelines when designing an authentication strategy:

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

Authorization

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

Consider the following guidelines when designing an authorization strategy:

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

Communication

When designing the communication strategy for your service, the protocol you choose should be based on the deployment scenario your service must support. If the service will be deployed within a closed network, you can use Transmission Control Protocol (TCP) for more efficient communications. If the service will be deployed into a public-facing network, you should choose the HyperText Transfer Protocol (HTTP) protocol.

Consider the following guidelines when designing a communication strategy:

- Determine how to handle unreliable or intermittent communication.
- Consider using dynamic URL behavior with configured endpoints for maximum flexibility.
- Validate endpoint addresses in messages.
- Determine whether you need to make asynchronous calls.
- Determine if you need request-response or duplex communication.
- Decide if message communication must be one-way or two-way.

Exception Management

Designing an effective exception-management strategy for your service layer is important for the security and reliability of your application. Failure to do so can make your application vulnerable to denial of service (DoS) attacks, and can also allow it to reveal sensitive and critical information.
Raising and handling exceptions is an expensive operation, so it is important for the design to take into account the potential impact on performance. A good approach is to design a centralized exception management and logging mechanism, and consider providing access points that support instrumentation and centralized monitoring in order to assist system administrators.

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

- Do not use exceptions to control business logic.
- Design a strategy for handling unhandled exceptions.
- Do not reveal sensitive information in exception messages or log files.

- Use SOAP Fault elements or custom extensions to return exception details to the caller.
- Disable tracing and debug-mode compilation for all services, except during development and testing.

Messaging Channels

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

Consider the following guidelines when designing message channels:

- Determine appropriate patterns for messaging channels, such as Channel Adapter, Messaging Bus, and Messaging Bridge.
- Determine how you will intercept and inspect the data between endpoints if necessary.

Message Construction

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

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

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

Message Endpoint

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

Consider the following guidelines when designing message endpoints:

- Determine relevant patterns for message endpoints, such as Gateway, Mapper, Competing Consumers, and Message Dispatcher.
- Determine if you should accept all messages or implement a filter to handle specific messages.

- Design for idempotency in your message interface. Idempotency is the situation where you could receive duplicate messages from the same consumer, but should only handle one. In other words, an idempotent endpoint will guarantee that only one message will be handled, and all duplicate messages will be ignored.
- Design for commutativity in your message interface. Commutativity is related to the order in which messages are received. In some cases, you may need to store inbound messages so that they can be processed in the correct order.
- Design for disconnected scenarios. For instance, you might need to support guaranteed delivery.

Message Protection

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

Consider the following guidelines when designing message protection:

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

Message Routing

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

Consider the following guidelines when designing message routing:

- Determine relevant patterns for message routing, such as Aggregator, Content-Based Router, Dynamic Router, and Message Filter.
- If sequential messages are sent from a consumer, the router must ensure that they are all delivered to the same endpoint in the required order (commutativity).

- A message router will normally inspect information in the message to determine how to route the message. As a result, you must ensure that the router can access that information.

Message Transformation

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

Consider the following guidelines when designing message transformation:

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

REST

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

Consider the following guidelines when designing REST resources:

- Identify and categorize resources that will be available to clients.
- Choose an approach for resource representation. A good practice would be to use meaningful names for REST starting points and unique identifiers, such as a globally unique identifier (GUID), for specific resource instances. For example, http://www.contoso.com/employee/ represents an employee starting point. http://www.contoso.com/employee/8ce762d5-b421-6123-a041-5fbd07321bac4 uses a GUID that represents a specific employee.
- Decide if multiple representations should be supported for different resources. For example, you can decide if the resource should support an XML, Atom, or JSON format and make it part of the resource request. A resource could be exposed as both http://www.contoso.com/example.atom and http://www.contoso.com/example.json
- Decide if multiple views should be supported for different resources. For example, decide if the resource should support GET and POST operations, or only GET operations.

Service Interface

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

Consider the following guidelines when designing a service interface:

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

SOAP

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

Consider the following guidelines when designing SOAP messages:

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

Deployment Considerations

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

Consider the following guidelines when deploying the service layer:

- Deploy the service layer to the same tier as the business layer to improve application performance, unless performance and security issues inherent within the production environment prevent this.
- If the service is located on the same physical tier as the service consumer, consider using the 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
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

Category	Relevant patterns
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.
- **Behavior (REST)**. Applies to resources that carry out operations. These resources generally contain no state of their own, and only support the POST operation.
- **Canonical Data Mapper**. Uses a common data format to perform translations between two disparate data formats.
- **Channel Adapter**. A component that can access the application’s API or data and publish messages on a channel based on this data, and that can receive messages and invoke functionality inside the application.
- **Claim Check**. Retrieves data from a persistent store when required.
- **Command Message**. Provides a message structure used to support commands.
- **Competing Consumer**. Sets multiple consumers on a single message queue and have them compete for the right to process the messages, which allows the messaging client to process multiple messages concurrently.
- **Container**. Builds on the entity pattern by providing the means to dynamically add and/or update nested resources.
- **Content Enricher**. Enriches messages with missing information obtained from an external data source.
- **Content Filter**. Removes sensitive data from a message and reduces network traffic by removing unnecessary data from a message.
- **Content-Based Router**. Routes each message to the correct consumer based on the contents of the message; such as existence of fields, specified field values, and so on.
- **Data Confidentiality**. Uses message-based encryption to protect sensitive data in a message.

- **Data Contract**. A schema that defines data structures passed with a service request.
- **Data Integrity**. Ensures that messages have not been tampered with in transit.
- **Data Origin Authentication**. Validates the origin of a message as an advanced form of data integrity.
- **Document Message**. A structure used to reliably transfer documents or a data structure between applications.
- **Duplex**. Two-way message communication where both the service and the client send messages to each other independently, irrespective of the use of the one-way or the request/reply pattern.
- **Durable Subscriber**. In a disconnected scenario, messages are saved and then made accessible to the client when connecting to the message channel in order to provide guaranteed delivery.
- **Dynamic Router**. A component that dynamically routes the message to a consumer after evaluating the conditions/rules that the consumer has specified.
- **Entity**. **(REST)**. Resources that can be read with a GET operation, but can only be changed by PUT and DELETE operations.
- **Envelope Wrapper**. A wrapper for messages that contains header information used, for example, to protect, route, or authenticate a message.
- **Event Message**. A structure that provides reliable asynchronous event notification between applications.
- **Exception Shielding**. Prevents a service from exposing information about its internal implementation when an exception occurs.
- **Façade**. Implements a unified interface to a set of operations in order to provide a simplified reduce coupling between systems.
- **Fault Contracts**. A schema that defines errors or faults that can be returned from a service request.
- **Federation**. An integrated view of information distributed across multiple services and consumers.
- **Fire and Forget**. A one-way message communication mechanism used when no response is expected.
- **Idempotent Receiver**. Ensures that a service will only handle a message once.
- **Message Broker (Hub-and-Spoke)**. A central component that communicates with multiple applications to receive messages from multiple sources, determines the correct destination, and route the message to the correct channel.
- **Message Bus**. Structures the connecting middleware between applications as a communication bus that enables the applications to work together using messaging.
- **Message Dispatcher**. A component that sends messages to multiple consumers.
- **Message Filter**. Eliminates undesired messages, based on a set of criteria, from being transmitted over a channel to a consumer.
- **Messaging Bridge**. A component that connects messaging systems and replicates messages between these systems.
- **Messaging Gateway**. Encapsulates message-based calls into a single interface in order to separate it from the rest of the application code.

- **Messaging Mapper**. Transforms requests into business objects for incoming messages, and reverses the process to convert business objects into response messages.
- **Normalizer**. Converts or transforms data into a common interchange format when organizations use different formats.
- **Point-to-point Channel**. Sends a message on a Point-to-Point Channel to ensure that only one receiver will receive a particular message.
- **Polling Consumer**. A service consumer that checks the channel for messages at regular intervals.
- **Process Manager**. A component that enables routing of messages through multiple steps in a workflow.
- **Publish-subscribe Channel**. Creates a mechanism to send messages only to the applications that are interested in receiving the messages without knowing the identity of the receivers.
- **Reliable Sessions**. End-to-end reliable transfer of messages between a source and a destination, regardless of the number or type of intermediaries that separate endpoints.
- **Remote Façade**. Creates a high-level unified interface to a set of operations or processes in a remote subsystem to make that subsystem easier to use, by providing a course-grained interface over fine-grained operations to minimize calls across the network.
- **Replay Protection**. Enforces message idempotency by preventing an attacker from intercepting a message and executing it multiple times.
- **Request Response**. A two-way message communication mechanism where the client expects to receive a response for every message sent.
- **Request-Reply**. Uses separate channels to send the request and reply.
- **Selective Consumer**. The service consumer uses filters to receive messages that match specific criteria.
- **Service Activator**. A service that receives asynchronous requests to invoke operations in business components.
- **Service Contract**. A schema that defines operations that the service can perform.
- **Service Interface**. A programmatic interface that other systems can use to interact with the service.
- **Store (REST)**. Allows entries to be created and updated with PUT.
- **Transaction (REST)**. Resources that support transactional operations.
- **Transactional Client**. A client that can implement transactions when interacting with a service.
- **Validation**. Checks the content and values in messages to protect a service from malformed or malicious content.

Technology Considerations

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

- Consider using ASP.NET Web services (ASMX) for simplicity, but only when a suitable Web server will be available.
- Consider using WCF services for advanced features and support for multiple transport protocols.

- If you are using ASMX and you require message-based security and binary data transfer, consider using Web Service Extensions (WSE).
- If you are using WCF and you want interoperability with non-WCF or non-Windows clients, consider using HTTP transport based on SOAP specifications.
- If you are using WCF 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 *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn.microsoft.com/en-us/library/ms998469.aspx.
- For more information, see *Web Service Security Guidance* at http://msdn.microsoft.com/en-us/library/aa480545.aspx
- For more information, see *Improving Web Services Security: Scenarios and Implementation Guidance for WCF* at http://www.codeplex.com/WCFSecurityGuide

PART IV

Archetypes

In This Part:

- ▶ Applications Archetypes
- ▶ Web Applications
- ▶ Rich Internet Applications (RIA)
- ▶ Rich Client Applications
- ▶ Services
- ▶ Mobile Applications
- ▶ Office Business Applications (OBA)
- ▶ SharePoint LOB Applications

Chapter 14: Application Archetypes

Objectives

- Understand the application types covered in this guide.
- Understand the tradeoffs necessary when choosing an application type.
- Understand the design impact when choosing an application type.
- Determine the appropriate application type for your scenario and requirements.

Overview

Your choice of application type will be determined by both the technology constraints you face and the type of user experience you plan to deliver. For example, you must decide whether the clients you intend to serve will have a permanent network connection available, whether you must deliver rich media content to anonymous users for viewing in a Web browser, or whether you will predominantly service a small number of users on a corporate intranet.

Use the following Application Types Summary to review each application type and its description. Use the table in the Choosing Application Types section to make an informed choice when choosing an application type, based on the benefits and considerations for each type. Use the Common Scenarios and Solutions section to map your application scenario to common application type solutions.

Application Types Summary

- **Mobile applications**. Applications of this type can be developed as thin client or rich client applications. Rich client mobile applications can support disconnected or occasionally connected scenarios. Web or thin client applications support connected scenarios only. The device resources may prove to be a constraint when designing mobile applications.
- **Rich client applications**. Applications of this type are usually developed as stand-alone applications with a graphical user interface that displays data using a range of controls. Rich client applications can be designed for disconnected and occasionally connected scenarios because the applications run on the client machine.
- **Rich Internet applications**. Applications of this type can be developed to support multiple platforms and multiple browsers, displaying rich media or graphical content. Rich Internet applications run in a browser sandbox that restricts access to some devices on the client.
- **Service applications**. Services expose complex functionality and allow clients to access them from local or remote machine. Service operations are called using messages, based on XML schemas, passed over a transport channel. The goal in this type of application is to achieve loose coupling between the client and the server.
- **Web applications**. Applications of this type typically support connected scenarios and can support different browsers running on a range of operating systems and platforms.

Mobile Application Archetype

A mobile application will normally be structured as a multilayered application consisting of user experience, business, and data layers, as shown in the following illustration.

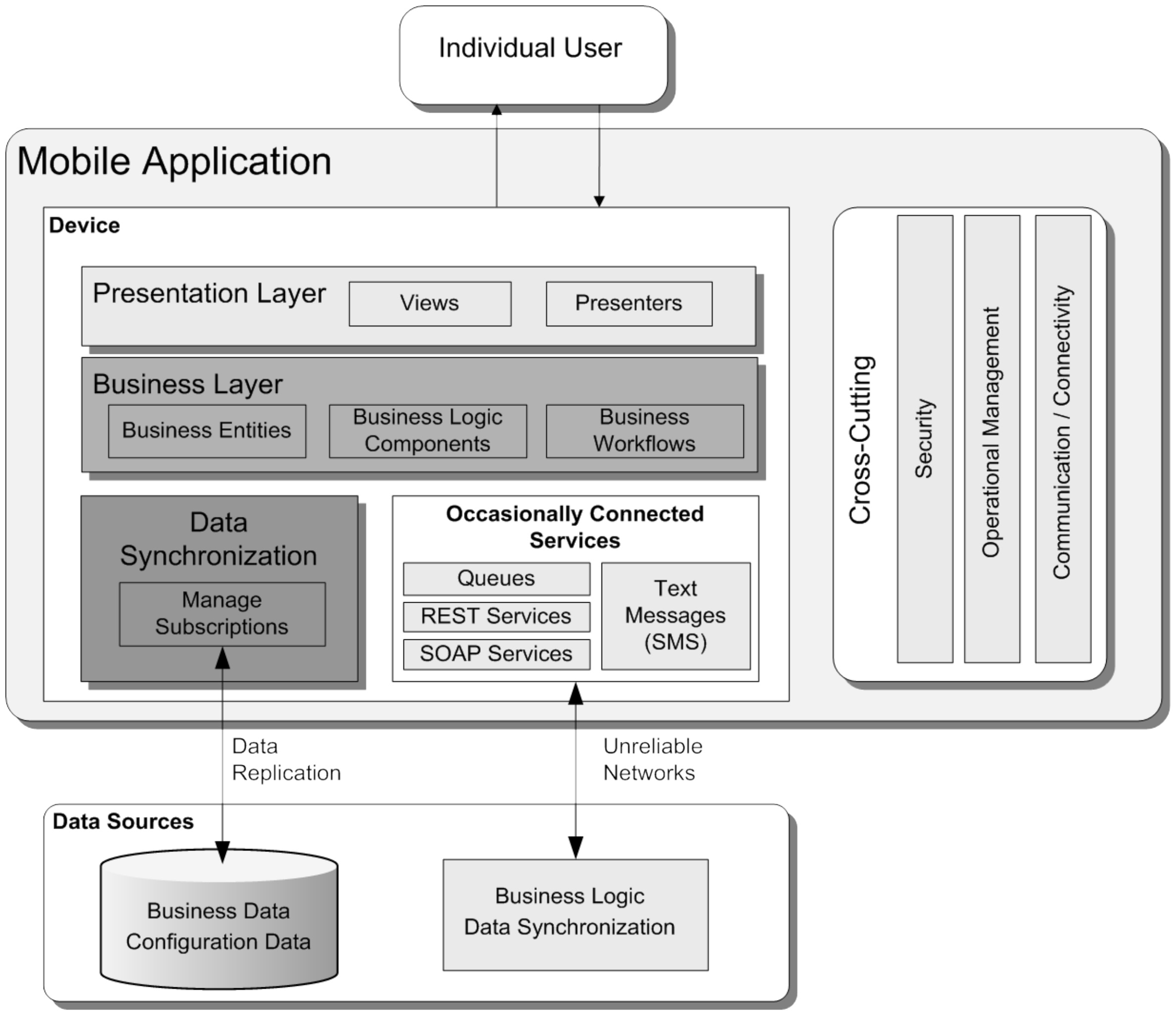

Figure 1 Mobile application archetype

When developing a mobile application, you may choose to develop a thin Web-based client or a rich client. If you are building a rich client, the business and data layers are likely to be on the device itself. If you are building a thin client, the business and data layers will be on the server.

Rich Client Application Archetype

Rich client user interfaces can provide high performance, improved interactivity, and a rich user experience for applications that must operate in stand-alone, connected, occasionally connected, and disconnected scenarios.

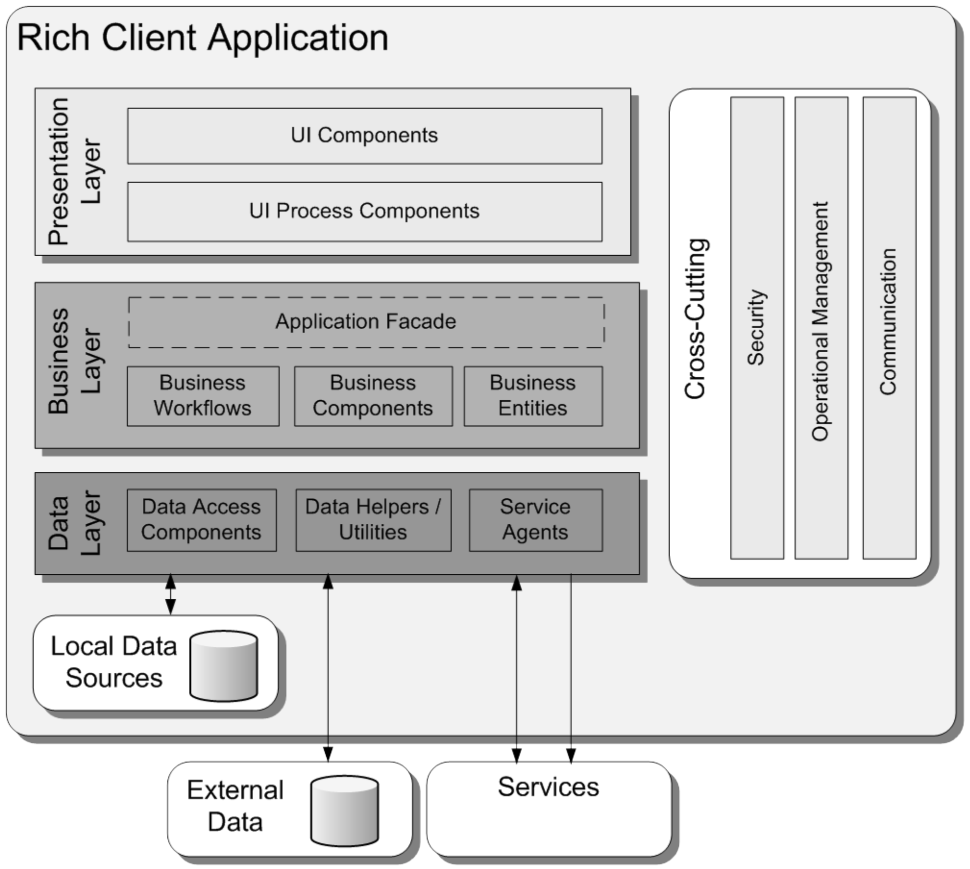

Figure 2 Rich client application

Rich Internet Application (RIA) Archetype

A rich Internet application (RIA) runs in the browser in a sandbox. The benefits of an RIA over traditional Web applications include a richer user experience, improved user responsiveness, and improved network efficiency.

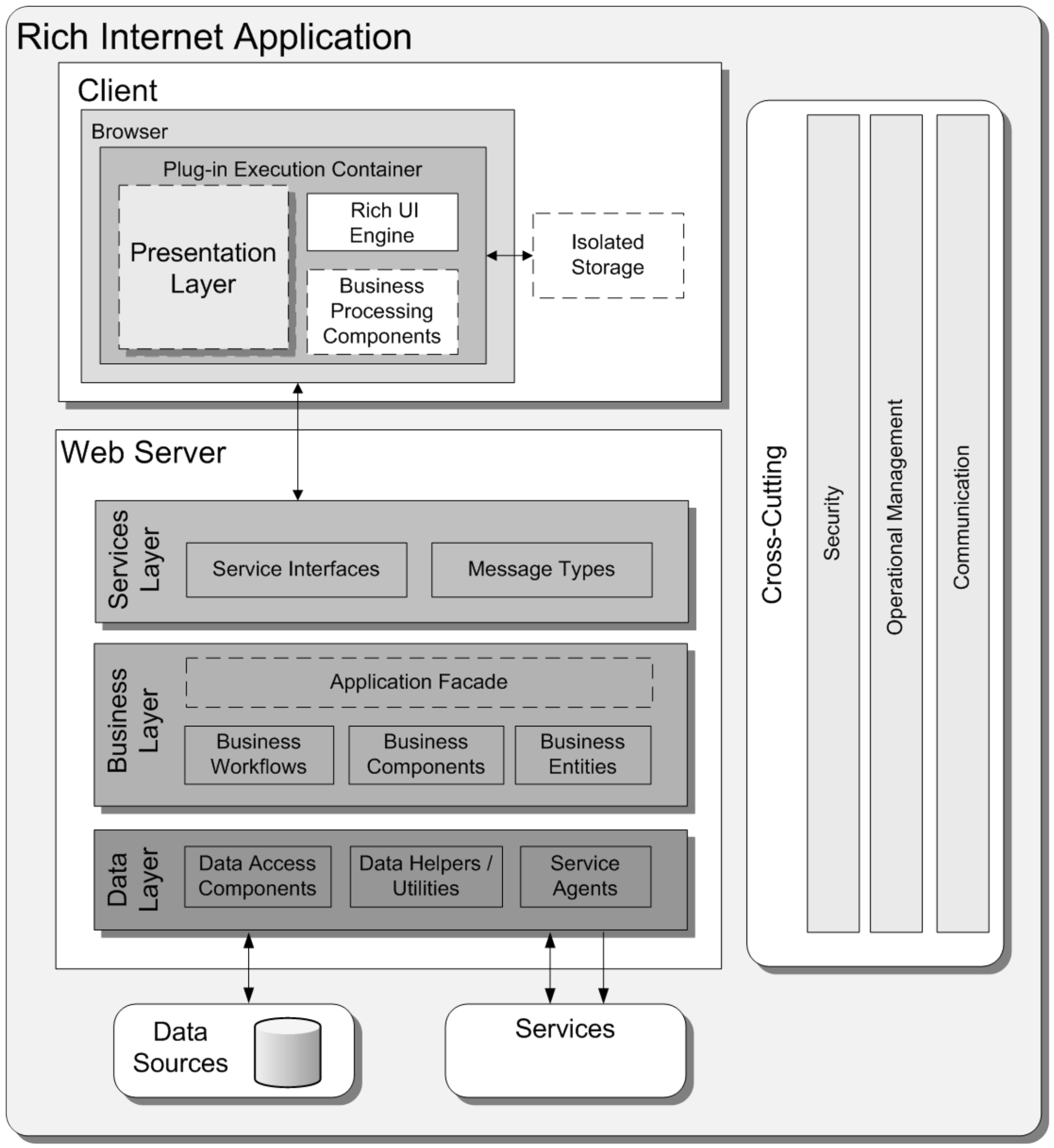

Figure 3 Rich Internet application

Service Archetype

In the context of this guide, a *service* is a public interface that provides access to a unit of functionality. Services literally provide some programmatic 'service' to the caller that consumes the service.

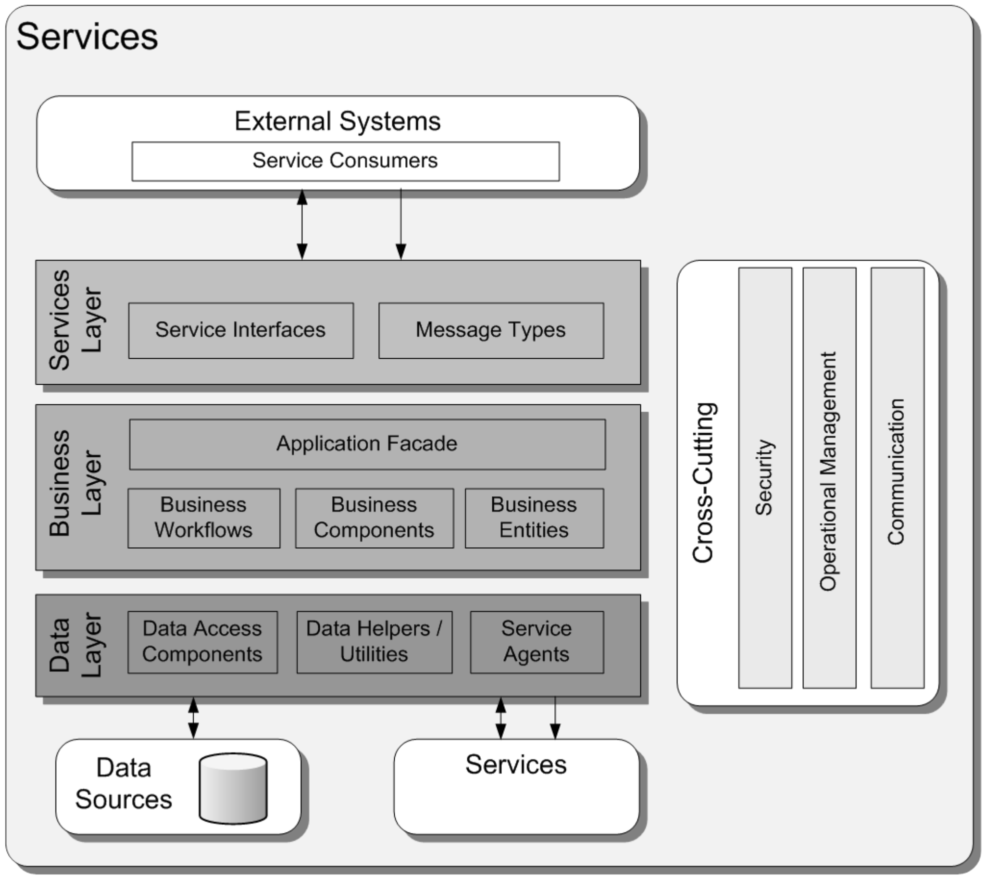

Figure 4 Service application

Services are loosely coupled, and can be combined within a client or within other services to provide functionality that is more complex. Services are distributable, and can be accessed from a remote machine as well as from the local machine on which the service is running. Services are also message oriented, meaning that service interfaces are defined by a Web Services Description Language (WSDL) document, and operations are called using messages based on XML schemas, which are passed over a transport channel. In addition, services support a heterogeneous environment by focusing interoperability on the message/interface definition. If components can understand the message and interface definition, they can use the service regardless of their base technology.

Web Application Archetype

The core of a Web application is its server-side logic. The Web application layer itself can be comprised of many distinct layers. A typical example is a three-layered architecture comprising presentation, business, and data layers.

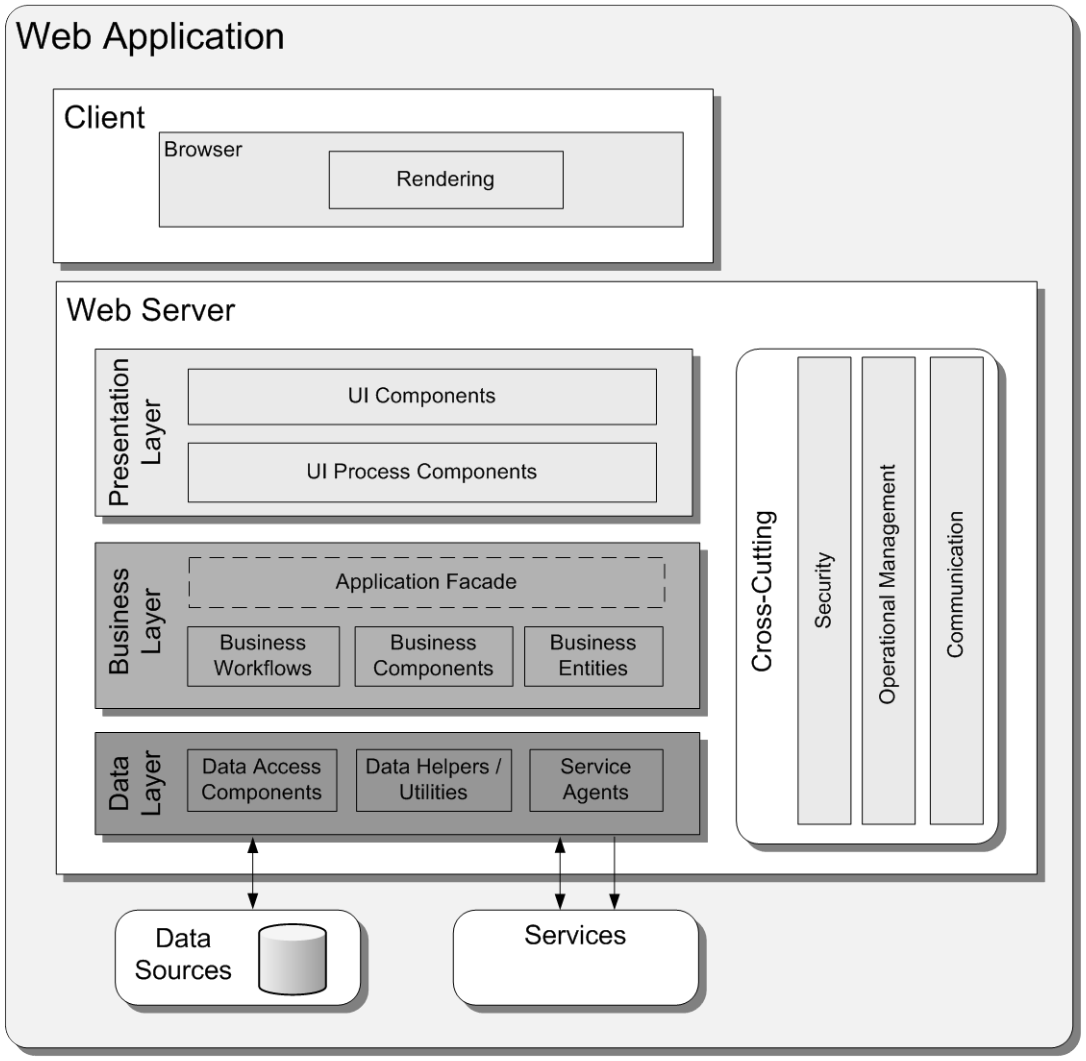

Figure 5 Web application

Choosing Application Types

Choose the appropriate application type by considering your specific requirements and infrastructure limitations. Use the Application Type Considerations table below to make an informed choice based on the benefits and considerations for each application type.

Application Type Considerations

Application type	Benefits	Considerations
Mobile Applications	• Support for handheld devices • Availability and ease of use for out-of-office users • Support for offline and occasionally-connected scenarios	• Input and navigation limitations • Limited screen display area
Rich client applications	• Ability to leverage client resources • Better responsiveness, rich UI functionality, and improved user experience • Highly dynamic and responsive interaction • Support for offline and occasionally connected applications	• Deployment complexity; however, a range of installation options such as ClickOnce, Windows Installer and XCOPY are available • Challenging to version over time • Platform-specific
Rich Internet applications (RIA)	• The same rich user interface capability as rich clients • Support for rich media and graphic display • Simple deployment and the distribution capabilities (reach) of Web clients • Simple upgrade and version updating • Cross-platform and cross-browser support	• Larger application footprint on the client machine compared to a Web application • Restrictions on leveraging client resources compared to a rich client application • Requirement for deployment of the .NET or Microsoft Silverlight™ run time on the client
Service applications	• Loosely coupled interactions between client and server • Ability to be consumed by different and unrelated applications • Support for interoperability	• No UI support • Dependent on network connectivity
Web applications	• Broad reach, and a standards-based UI across multiple platforms • Ease of deployment and change management	• Dependent on continual network connectivity • Difficulty in providing a rich user interface

Common Scenarios and Solutions

Your choice of application type will be driven primarily by the scenarios that you want to support. Use the following scenarios to help understand which application type is the best fit for your needs.

- **Mobile applications** are suited for scenarios in which you want to support mobile device users.

- **Rich client applications** are suited for scenarios in which you want to leverage client resources and support disconnected scenarios.
- **Rich Internet applications** are suited for scenarios in which you want to support advanced graphics and media in a Web-deployed application.
- **Service applications** are suited for scenarios in which you want to expose a loosely coupled interface to remote clients without a UI.
- **Web applications** are suited for scenarios in which you want to provide the application's UI over the Web.

Mobile Applications

Consider using mobile applications if:

- Your users depend on handheld devices.
- Your application supports a simple UI that is suitable for use on small screens.
- Your application must support offline or occasionally connected scenarios; consider designing a mobile rich client.
- Your application must be device-independent and can depend on network connectivity; consider designing a mobile Web client.

Rich Client Applications

Consider using rich client applications if:

- Your application must support disconnected or occasionally connected scenarios.
- Your application will be deployed on client PCs.
- Your application must be highly interactive and responsive.
- Your application UI must provide rich functionality and user interaction but doesn't need the advanced graphics or media capabilities of a RIA.
- Your applications must utilize the resources of the client PC.

Rich Internet Applications

Consider using rich Internet applications if:

- Your application must support rich media and provide a highly graphical display.
- Your application must provide a rich, interactive, and responsive UI compared to Web applications.
- Your application will leverage client-side processing in a restricted manner.
- Your application will utilize client-side resources in a restricted manner.
- You want the simplicity of a Web-based deployment model.

Service Applications

Consider using service applications if:

- Your application will expose functionality that does not need UI support.
- Your application must be loosely coupled with its clients.
- Your application must be shared with or consumed by other external applications.

- Your application must expose functionality that will be consumed by applications over the Internet, an intranet, and on the local machine.

Web Applications

Consider using Web applications if:

- Your application does not require the rich UI and media support offered by a rich Internet application.
- You want the simplicity of a Web-based deployment model.
- Your application must be platform-independent.
- Your application must be available over the Internet.
- You want to minimize client-side dependencies and impact, such as disk or processor usage.

Technology Considerations

Each application type is supported by one or more technologies that can be used to implement your application. Your choice of technology will be driven by scenarios and technology constraints, as well as the capabilities and experience of your development team.

Mobile Applications

The following presentation-layer technologies are available for creating mobile applications:

- **.NET Compact Framework**. You can use the .NET Compact Framework to create a rich client mobile application that supports connected or occasionally connected scenarios.
- **ASP.NET Mobile**. You can use ASP.NET mobile controls to create a thin client mobile application. ASP.NET mobile controls are a set of server-side controls and special page classes that render output specific to the browser making the request.
- **Silverlight**. You can use Silverlight for mobile devices to provide rich media support and an improved user experience.

Rich Client Applications

The following presentation-layer technologies are available for creating rich client applications:

- **Windows Forms**. You can use Windows Forms to create applications that provide rich functionality and user experience by utilizing the resources of the client PC.
- **Windows Forms with WPF user controls**. You can use WPF user controls in Windows Forms applications to provide enhanced rich graphical support within the user interface.
- **WPF**. You can use WPF to create a rich client application with UI support for 2-D and 3-D graphics, and for animations and media (both video and audio). WPF also includes a two-way data-binding engine.
- **XAML Browser Application (XBAP) using WPF**. You can create an XBAP that provides all the features of the stand-alone WPF application, but is hosted in a browser.

Rich Internet Applications (RIA)

The following presentation-layer technologies are available for creating rich Internet applications:

- **Silverlight**. You can use Silverlight to create applications that provide a rich user experience that includes graphics, audio, and video.
- **Silverlight with AJAX**. You can combine Silverlight with Asynchronous JavaScript and XML (AJAX) to create a rich Internet application that performs asynchronous communication between the client and the server.

Service Applications

The following technologies are available for creating service applications:

- **Windows Communication Foundation (WCF)**. When possible, use WCF to create services in order to benefit from maximum feature availability and interoperability.
- **ASP.NET Web services (ASMX)**. Use ASMX for simplicity and when it's ok to host your server from an IIS web server.

Web Applications

The following presentation-layer technologies are available for creating Web applications:

- **ASP.NET Web Forms**. You can use ASP.NET Web Forms with a wide range of server controls that render HTML in Web browsers.
- **ASP.NET Web Forms with AJAX**. You can use AJAX in your ASP.NET Web Forms application to improve the user experience by reducing the number of post backs required.
- **ASP.NET Web Forms with Silverlight controls**. You can use Silverlight controls in your ASP.NET Web application to provide a rich user experience and support media streaming.
- **ASP.NET Model-View-Controller (MVC)**. You can use ASP.NET MVC to create Web applications with built-in support for the Model-View-Controller design pattern. MVC simplifies developing, modifying, and testing the individual components within the application.
- **ASP.NET Dynamic Data**. You can use ASP.NET Dynamic Data to create functional data-driven Web applications based on a LINQ to SQL or Entity Framework data model.

Chapter 15: Web Application Archetype

Objectives

- Define a Web application.
- Learn the general design considerations for a Web application.
- Learn the guidelines for key attributes of a Web application.
- Learn the guidelines for layers within a Web application.
- Learn the guidelines for performance, security, and deployment.
- Learn the key patterns and technology considerations.

Overview

The core of a Web application is its server-side logic. The Web application layer itself can be comprised of many distinct layers. The typical example is a three-layered architecture comprised of presentation, business, and data layers. Figure 1 illustrates a common Web application architecture with common components grouped by different areas of concern.

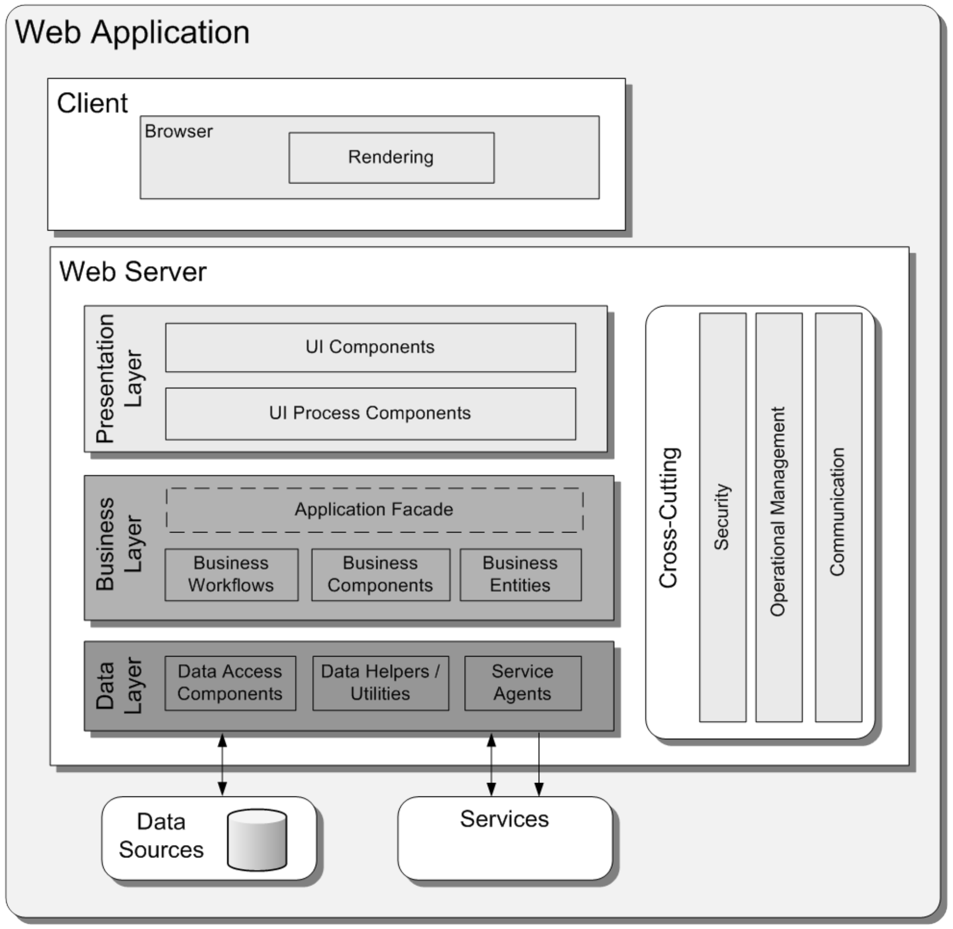

Figure 1 A common Web application architecture

Design Considerations

When designing a Web application, the goals of a software architect are to minimize the complexity by separating tasks into different areas of concern while designing a secure, high-performance application.

When designing your Web application, consider the following guidelines:

- **Partition your application logically.** Use layering to partition your application logically into presentation, business, and data access layers. This helps you to create maintainable code and allows you to monitor and optimize the performance of each layer separately. A clear logical separation also offers more choices for scaling your application.
- **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 shared abstraction that must be implemented by interface components.
- **Understand how components will communicate with each other**. This requires an understanding of the deployment scenarios your application must support. You must determine if communication across physical boundaries or process boundaries should be supported, or if all components will run within the same process.
- **Reduce round trips.** When designing a Web application, consider using techniques such as caching and output buffering to reduce round trips between the browser and the Web server, and between the Web server and downstream servers.
- **Consider using caching.** A well-designed caching strategy is probably the single most important performance-related design consideration. ASP.NET caching features include output caching, partial page caching, and the cache API. Design your application to take advantage of these features.
- **Consider using logging and instrumentation.** You should audit and log activities across the layers and tiers of your application. These logs can be used to detect suspicious activity, which frequently provides early indications of an attack on the system.
- **Avoid blocking during long-running tasks.** If you have long-running or blocking operations, consider using an asynchronous approach to allow the Web server to process other incoming requests.
- **Consider authenticating users across trust boundaries.** You should design your application to authenticate users whenever they cross a trust boundary; for example, when accessing a remote business layer from your presentation layer.
- **Do not pass sensitive data in plain text across the network.** Whenever you need to pass sensitive data such as a password or authentication cookie across the network, consider encrypting and signing the data or using Secure Sockets Layer (SSL) encryption.
- **Design your Web application to run using a least-privileged account.** If an attacker manages to take control of a process, the process identity should have restricted access to the file system and other system resources in order to limit the possible damage.

Web Application Frame

There are several common issues that you must consider as you 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
Authentication	• Lack of authentication across trust boundaries • Storing passwords in a database as plain text • Designing custom authentication mechanism instead of using built-in capabilities
Authorization	• Lack of authorization across trust boundaries • Incorrect role granularity • Using impersonation and delegation when not required
Caching	• Caching volatile data • Not considering caching page output • Caching sensitive data • Failing to cache data in a ready-to-use format
Exception Management	• Revealing sensitive information to the end user • Not logging sufficient details about the exception • Using exceptions to control application flow
Logging and Instrumentation	• Failing to implement adequate instrumentation in all layers • Failing to log system-critical and business-critical events • Not supporting run-time configuration of logging and instrumentation • Logging sensitive information
Navigation	• Mixing navigation logic with user interface components • Hard-coding relationships between views • Not verifying if the user is authorized to navigate to a view
Page Layout (UI)	• Using table-based layout for complex layouts • Designing complex and overloaded pages
Page Rendering	• Using post backs and page refreshes for many user interactions • Using excessive page sizes that reduce performance
Presentation Entity	• Creating custom entity objects when not required • Adding business logic to presentation entities
Request Processing	• Mixing processing and rendering logic • Choosing an inappropriate pattern
Service Interface	• Breaking the service interface • Implementing business rules in a service interface • Failing to consider interoperability requirements
Session Management	• Using an incorrect session store • Not considering serialization requirements • Not persisting session data when required
Validation	• Failure to implement server-side validation

Category	Key issues
	• Lack of validation across trust boundaries • Not reusing the validation logic

Authentication

Designing an effective authentication strategy is important for the security and reliability of your application. Improper or weak authorization can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attack.

Consider the following guidelines when designing an authentication strategy:

- Identify trust boundaries within Web application layers. This will help you to determine where to authenticate.
- Use a platform-supported authentication mechanism such as Windows Authentication when possible.
- If you are using Forms Authentication, use the platform features, such as the forms element, when possible.
- Enforce strong account management practices such as account lockouts and expirations.
- Enforce strong password policies. This includes specifying password length and complexity, and password expiration policies.

Authorization

Authorization determines the tasks that an authenticated identity can perform, and identifies the resources that can be accessed. Designing an effective authorization strategy is important for the security and reliability of your application. Improper or weak authorization leads to information disclosure, data tampering, and elevation of privileges. Defense in depth is the key security principle to apply to your application's authorization strategy.

Consider the following guidelines when designing an authorization strategy:

- Identify trust boundaries within the Web application layers and authorize users across trust boundaries.
- Use URL authorization for page and directory access control.
- Consider the granularity of your authorization settings. Building your authorization with too much granularity will increase your management overhead; however, using too much granularity will reduce flexibility.
- Access downstream resources using a trusted identity based on the trusted subsystem model.
- Use impersonation and delegation to take advantage of the user-specific auditing and granular access controls of the platform, but consider the effect on performance and scalability.

Caching

Caching improves the performance and responsiveness of your application. However, incorrect caching choices and poor caching design 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 first decide when to load data into the cache. Try to load cache data asynchronously or by using a batch process to avoid client delays.

Consider the following guidelines when designing caching:

- Avoid caching volatile data.
- Use output caching to cache pages that are relatively static.
- Consider using partial page caching through user controls for static data in your pages.
- Pool shared resources that are expensive, such as network connections, instead of caching them.
- Cache data in a ready-to-use format.

Exception Management

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

Consider the following guidelines when designing an exception management strategy:

- Do not use exceptions to control the logical flow of your application.
- Do not catch exceptions unless you must handle them, you need to strip sensitive information, or you need to add additional information to the exception.
- Design a global error handler to catch unhandled exceptions.
- Display user-friendly messages to end users whenever an error or exception occurs.
- Do not reveal sensitive information, such as passwords, through exception details.

Logging and Instrumentation

Designing an effective logging and instrumentation strategy is important for the security and reliability of your application. You should audit and log activity across the tiers of your application. These logs can be used to detect suspicious activity, which frequently provides early indications of an attack on the system, and help to address the repudiation threat where users deny their actions. Log files may be required in legal proceedings to prove the wrongdoing of individuals. Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access and by the same routines that access the resource.

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

- Consider auditing for user management events.

- Consider auditing for unusual activities.
- Consider auditing for business-critical operations.
- Create secure log file management policies, such as restricting the access to log files, allowing only write access to users, etc.
- Do not store sensitive information in the log or audit files.

Navigation

Design your navigation strategy in a way that separates it from the processing logic. Your strategy should allow users to navigate easily through your screens or pages. Designing a consistent navigation structure for your application will help to minimize user confusion as well as reduce the apparent complexity of the application.

Consider the following guidelines when designing your navigation strategy:

- Use well-known design patterns, such as Model-View-Presenter (MVP), to decouple UI processing from output rendering.
- Consider encapsulating navigation in a master page so that it is consistent across pages.
- Design a site map to help users find pages on the site, and to allow search engines to crawl the site if desired.
- Consider using wizards to implement navigation between forms in a predictable way.
- Consider using visual elements such as embedded links, navigation menus, and breadcrumb navigation in the UI to help users understand where they are, what is available on the site, and how to navigate the site quickly.

Page Layout (UI)

Design your application so that the page layout can be separated from the specific UI components and UI processing. When choosing a layout strategy, consider whether designers or developers will be building the layout. If designers will be building the layout, choose a layout approach that does not require coding or the use of development-focused tools.

Consider the following guidelines when designing your layout strategy:

- Use Cascading Style Sheets (CSS) for layout whenever possible.
- Use table-based layout when you need to support a grid layout, but remember that table-based layout can be slow to render, does not have full cross-browser support, and there may be issues with complex layout.
- Use a common layout for pages where possible to maximize accessibility and ease of use.
- Use master pages in ASP.NET applications to provide a common look and feel for all of the pages.
- Avoid designing and developing large pages that accomplish multiple tasks, particularly where only a few tasks are usually executed with each request.

Page Rendering

When designing for page rendering, you must ensure that you render the pages efficiently and maximize interface usability.

Consider the following guidelines when designing a page-rendering strategy:

- Consider data-binding options. For example, you can bind custom objects or datasets to controls. However, be aware that binding only applies to rendered data in ASP.NET.
- Consider using Asynchronous JavaScript and XML (AJAX) for an improved user experience and better responsiveness.
- Consider using data-paging techniques for large amounts of data to minimize scalability issues.
- Consider designing to support localization in UI components.
- Abstract the user process components from data rendering and acquisition functions.

Presentation Entity

Presentation entities store the data that you will use to manage the views in your presentation layer. Presentation entities are not always necessary. Consider using presentation entities only if the datasets are sufficiently large or complex that they must be stored separately from the UI controls. Design or choose appropriate presentation entities that you can easily bind to UI controls.

Consider the following guidelines when designing presentation entities:

- Determine if you need presentation entities. Typically, you might need presentation entities if the data or data format to be displayed is specific to the presentation layer.
- Consider the serialization requirements for your presentation entities, if they are to be passed across the network or stored on the disk.
- Consider implementing data type validation in the property setters of your presentation entities.
- Consider using presentation entities to store state related to the UI. If you want to use this state to help your application recover from a crash, make sure after recovery that the user interface is in a consistent state.

Request Processing

When designing a request-processing strategy, you should ensure separation of concerns by implementing the request-processing logic separately from the UI.

Consider the following guidelines when designing a request-processing strategy:

- Consider centralizing the common pre-processing and post-processing steps of Web page requests to promote logic reuse across pages. For example, consider creating a base class derived from the ASP.NET **Page** class to contain your common pre- and post-processing logic.

- Consider dividing UI processing into three distinct roles—model, view, and controller/presenter—by using the Model-View-Controller (MVC) or Model-View-Presenter (MVP) pattern.
- If you are designing views for handling large amounts of data, consider giving access to the model from the view by using the Supervising Controller pattern, which is a form of the MVP pattern.
- If your application does not have a dependency on view state and you have a limited number of control events, consider using the MVC pattern.
- Consider using the Intercepting Filter pattern to implement the processing steps as pluggable filters when appropriate.

Session Management

When designing a Web application, an efficient and secure session-management strategy is important for performance and reliability. You must consider session-management factors such as what to store, where to store it, and how long information will be kept.

Consider the following guidelines when designing a session-management strategy:

- If you have a single Web server, require optimum session state performance, and have a relatively limited number of concurrent sessions, use the in-process state store.
- If you have a single Web server, your sessions are expensive to rebuild, and you require durability in the event of an ASP.NET restart, use the session state service running on the local Web server.
- Use a remote session state service or the Microsoft SQL Server® state store for Web farm scenarios.
- If you are storing state on a separate server, protect your session state communication channel.
- Prefer basic types for session data to reduce serialization costs.

Validation

Designing an effective validation solution is important for the security and reliability of your application. Improper or weak authorization can leave your application vulnerable to cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attack.

Consider the following guidelines when designing a validation strategy:

- Identify trust boundaries within Web application layers, and validate all data crossing these boundaries.
- Assume that all client-controlled data is malicious and needs to be validated.
- Design your validation strategy to constrain, reject, and sanitize malicious input.
- Design to validate input for length, range, format, and type.
- Use client-side validation for user experience, and server-side validation for security.

Presentation Layer Considerations

The presentation layer of your Web application displays the UI and facilitates user interaction. The design should focus on separation of concerns, where the user interaction logic is decoupled from the UI components.

Consider the following guidelines when designing the presentation layer:

- Consider separating the UI components from the UI process components.
- Use client-side validation to improve user experience and responsiveness, and server-side validation for security. Do not rely on just client-side validation.
- Use page output caching or fragment caching to cache static pages or parts of pages.
- Use Web server controls if you need to compile these controls into an assembly for reuse across applications, or if you need to add additional features to existing server controls.
- Use Web user controls if you need to reuse UI fragments on several pages, or if you want to cache a specific parts of the page.

Business Layer Considerations

When designing the business layer for your Web application, consider how to implement the business logic and long-running workflows. Design business entities that represent the real world data, and use these to pass data between components.

Consider the following guidelines when designing the business layer:

- Design a separate business layer that implements the business logic and workflows. This improves the maintainability and testability of your application.
- Consider centralizing and reusing common business logic functions.
- Design your business layer to be stateless. This helps to reduce resource contention and increase performance.
- Use a message-based interface for the business layer. This works well with a stateless Web application business layer.
- Design transactions for business-critical operations.

Data Layer Considerations

Design a data layer for your Web application to abstract the logic necessary to access the database. Using a separate data layer makes the application easier to configure and maintain. The data layer may also need to access external services using service agents.

Consider the following guidelines when designing the data layer:

- Design a separate data layer to hide the details of the database from other layers of the application.
- Design entity objects to interact with other layers, and to pass the data between them.
- Design to take advantage of connection pooling to minimize the number of open connections.

- Design an exception-handling strategy to handle data access errors, and to propagate exceptions to business layers.
- Consider using batch operations to reduce round trips to the database.

Service Layer Considerations

Consider designing a separate service layer if you plan to deploy your business layer on a remote tier, or if you plan to expose your business logic using a Web service.

Consider the following guidelines when designing the service layer:

- If your business layer is on a remote tier, design coarse-grained service methods to minimize the number of client-server interactions, and to provide loose coupling.
- Design the services without assuming a specific client type.
- Design the services to be idempotent, assuming that the same message request may arrive multiple times.

Testing and Testability Considerations

Testability is a measure of how well your system or components allow you to create test criteria and execute tests to determine if the criteria are met. You should consider testability when designing your architecture because it makes it easier to diagnose problems earlier and reduce maintenance cost. To improve the testability of your application, you can use logging events, provide monitoring resources, and implement test interfaces.

Consider the following guidelines for testability:

- Clearly define the inputs and outputs of the application or components during the design phase.
- Consider using the Passive View pattern (a variation of the MVP pattern) in the presentation layer, which removes the dependency between the view and the model.
- Design a separate business layer to implement the business logic and workflows, which improves the testability of your application.
- Design an effective logging strategy, which allows you to detect bugs that might otherwise be difficult to discover. Logging will help you to focus on faulty code when bugs are found. Log files should contain information that can be used to replicate the issues.
- Design loosely coupled components that can be tested individually.

Performance Considerations

You should identify your performance objectives early in the design phase of a Web application by gathering the non-functional requirements. Response time, throughput, CPU, memory, and disk I/O are a few of the key factors you should consider when designing your application.

Consider the following guidelines for performance:

- Ensure that the performance requirements are specific, realistic, and flexible.

- Implement caching techniques to improve the performance and scalability of the application.
- Perform batch operations to minimize round trips across boundaries.
- Reduce the volume of HTML transferred between server and client. For instance, you can disable view state when you do not need it; limit the use of graphics, and considering using compressed graphics where appropriate.
- Avoid unnecessary round trips over the network.

Security Considerations

Security is an important consideration for protecting the integrity and privacy of the data and the resources of your Web application. You should design a security strategy for your Web application that uses tested and proven security solutions, and implement authentication, authorization, and data validation to protect your application from a range of threats.

Consider the following guidelines for security:

- Consider the use of authentication at every trust boundary.
- Consider implementing a strong authorization mechanism to restrict resource access and protect business logic.
- Consider the use of input validation and data validation at every trust boundary to mitigate security threats such as cross-site scripting and code-injection.
- Do not rely on client-side validation only. Use server-side validation as well.
- Consider encrypting and digitally signing any sensitive data that is sent across the network.

Deployment Considerations

When deploying a Web application, you should take into account how layer and component location will affect the performance, scalability, and security of the application. You might also need to consider design trade-offs. Use either a distributed or a non-distributed deployment approach, depending on the business requirements and infrastructure constraints.

Consider the following guidelines for deployment:

- Consider using non-distributed deployment to maximize performance.
- Consider using distributed deployment to achieve better scalability and to allow each layer to be secured separately.

Non-Distributed Deployment

In a non-distributed deployment scenario, all the logically separate layers of the Web application are physically located on the same Web server, except for the database. You must consider how the application will handle multiple concurrent users, and how to secure the layers that reside on the same server. Figure 2 shows this scenario.

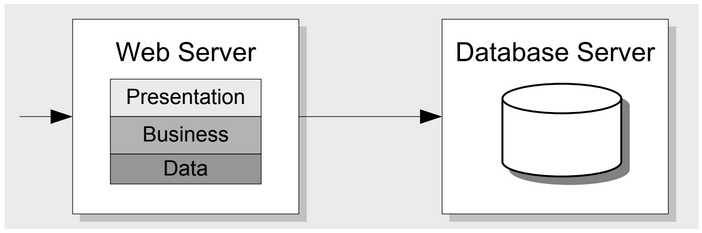

Figure 2 Non-distributed deployment of a Web application

Consider the following guidelines when choosing a non-distributed deployment:

- Consider using non-distributed deployment if your Web application is performance-sensitive, because the local calls to other layers provide performance gains.
- Consider designing a component-based interface for your business layer.
- If your business logic runs in the same process, avoid authentication at the business layer.
- Consider using a trusted identity (through the trusted subsystem model) to access the database. This improves the performance and scalability of your application.
- Consider encrypting and digitally signing sensitive data passed between the Web server and database server.

Distributed Deployment

In a distributed deployment scenario, the presentation and business layers of the Web application reside on separate physical tiers, and communicate remotely. You will typically locate your business and data access layers on the same sever. Figure 3 shows this scenario.

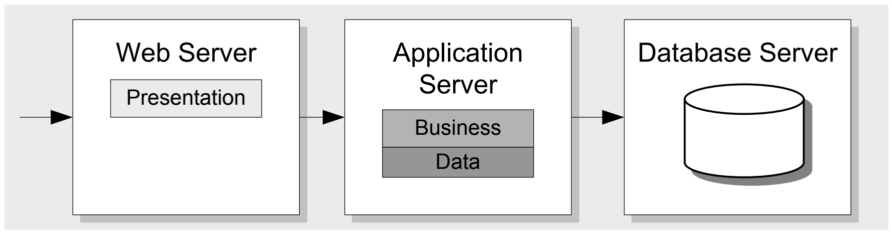

Figure 3 Distributed deployment of a Web application

Consider the following guidelines when choosing a distributed deployment:

- Do not physically separate your business logic components unless this is necessary.
- If your security concerns prohibit you from deploying your business logic on your front-end Web server, consider distributed deployment.
- Consider using a message-based interface for your business layer.
- Consider using the TCP protocol with binary encoding to communicate with the business layer for best performance.
- Consider protecting sensitive data passed between different physical tiers.

Load Balancing

When you deploy your Web application on multiple servers, you can use load balancing to distribute requests so that they are handled by different Web servers. This helps to maximize response times, resource utilization, and throughput. Figure 4 shows this scenario.

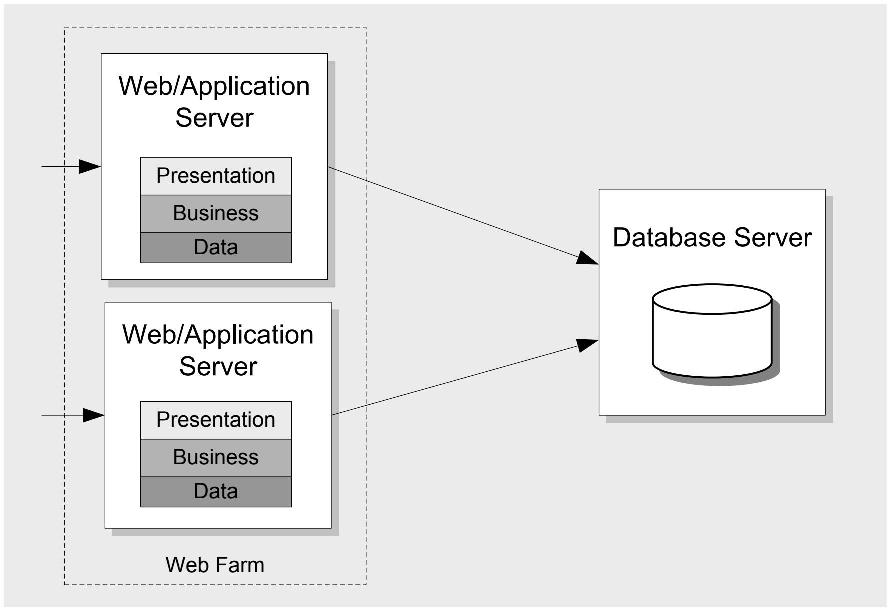

Figure 4 Load balancing a Web application

Consider the following guidelines when designing your Web application to use load balancing:

- Avoid server affinity when designing scalable Web applications. Server affinity occurs when all requests from a particular client must be handled by the same server. It usually occurs when you use locally updatable caches, or in-process or local session state stores.
- Consider designing stateless components for your Web application; for example, a Web front end that has no in-process state and no stateful business components.
- Consider using Windows Network Load Balancing (NLB) as a software solution to implement redirection of requests to the servers in an application farm.

Web Farm Considerations

A Web farm allows you to scale out your application, which can also minimize the impact of hardware failures. When you add more servers, you can use either a load-balancing or clustering approach.

Consider the following guidelines when designing your Web application to use a Web farm:

- Consider using clustering to minimize the impact of hardware failures.
- Consider partitioning your database across multiple database servers if your application has high input/output requirements.
- Consider configuring the Web farm to route all requests from the same user to the same server in order to provide affinity where this is required.
- Do not use in-process session management in a Web farm when requests from the same user cannot be guaranteed to be routed to the same server. Use an out-of-process state server service or a database server for this scenario.

Pattern Map

Category	Relevant patterns
Caching	• Cache Dependency • Page Cache
Exception Management	• Exception Shielding
Logging and Instrumentation	• Provider
Navigation	• Model-View-Presenter • Model-View-Controller
Page Layout (UI)	• Template View • Composite View • Transform View • Two-Step View
Request Processing	• Intercepting Filter • Page Controller • Front Controller • Passive View • Supervising Controller
Service Interface Layer	• Façade • Service Interface

Pattern Descriptions

- **Cache Dependency.** Use external information to determine the state of data stored in a cache.
- **Composite View.** Combine individual views into a composite representation.
- **Exception Shielding.** Filter exception data that should not be exposed to external systems or users.
- **Façade.** Implement a unified interface to a set of operations to provide a simplified, reduced coupling between systems.
- **Front Controller.** Consolidate request handling by channeling all requests through a single handler object, which can be modified at run time with decorators.
- **Intercepting Filter.** Create a chain of composable filters (independent modules) to implement common pre-processing and post-processing tasks during a Web page request.
- **Model-View-Controller.** Separate the UI code into three separate units: Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Model-View-Presenter.** Separate request processing into three separate roles, with the View being responsible for handling user input and passing control to a **Presenter** object.
- **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.
- **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**.**
- **Supervising Controller.** A variation of the MVC pattern in which the controller handles complex logic, in particular coordinating between views, but where 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 to be displayed on the UI.
- **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.

Technology Considerations

On the Microsoft platform, from an ASP.NET standpoint, you can combine the ASP.NET Web Forms model with a range of other technologies, including ASP.NET AJAX, ASP.NET MVC, Microsoft Silverlight™, and ASP.NET Dynamic Data. Consider the following guidelines:

- If you want to build applications that are accessed through a Web browser or specialized user agent, consider using ASP.NET.
- If you want to build applications that provide increased interactivity and background processing, with fewer page reloads, consider using ASP.NET with AJAX.
- If you want to build applications that include rich media content and interactivity, consider using ASP.NET with Silverlight controls.
- If you are using ASP.NET, consider using master pages to implement a consistent UI across all pages.
- 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.

Additional Resources

- For more information on designing and implementing Web client applications, see *Design and Implementation Guidelines for Web Clients* at http://msdn.microsoft.com/en-us/library/ms978605.aspx.
- For more information on designing distributed Web applications, see *Designing Distributed Applications* at http://msdn.microsoft.com/en-us/library/aa292470(VS.71).aspx.
- For more information on Web application performance issues, see *Improving .NET Application Performance and Scalability* at http://msdn.microsoft.com/en-us/library/ms998530.aspx.

- For more information on Web application security, see *Improving Web Application Security: Threats and Countermeasures* at http://msdn.microsoft.com/en-us/library/ms994921.aspx.

Chapter 16: Rich Internet Applications

Objectives

- Define a rich Internet application (RIA).
- Understand key scenarios where RIAs would be used.
- Understand the components found in an RIA.
- Learn the design considerations for RIAs.
- Learn the guidelines for performance, security, and deployment of RIAs.
- Learn the key patterns and technology considerations for designing RIAs.

Overview

Rich Internet applications (RIAs) support rich graphics and streaming media scenarios, while providing most of the deployment and maintainability benefits of a Web application. RIAs run in a browser plug-in, such as Microsoft® Silverlight®, as opposed to extensions that utilize browser code, such as Asynchronous JavaScript and XML (AJAX). A typical RIA implementation utilizes a Web infrastructure combined with a client-side application that handles the presentation. The plug-in provides library routines for rich graphics support as well as a container limiting access to local resources for security purposes. RIAs have the ability to run more extensive and complex client-side code than possible in a normal Web application, thus providing the opportunity to reduce load on the Web server.

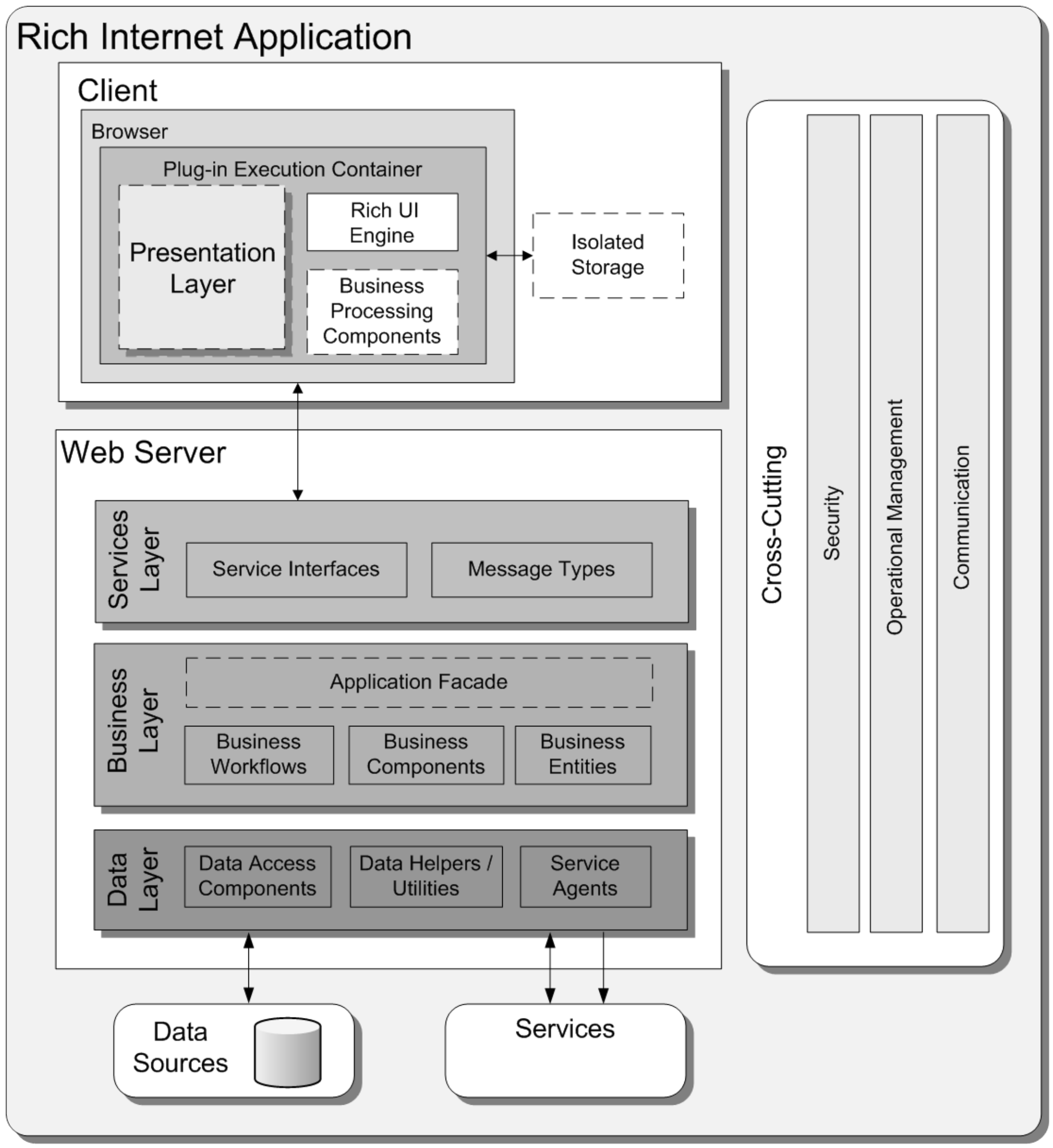

Figure 1 Architecture of a typical RIA implementation

Design Considerations

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

- **Choose an RIA based on audience, rich interface, and ease of deployment**. Consider designing an RIA when your vital audience is using a browser and operating system that supports RIAs. If part of your vital audience is on a non-RIA-supported browser, consider whether you can influence limiting browser choice to a supported version. If you cannot influence the browser choice, consider if the loss of audience is significant enough to require choosing another application type, such as a Web application using AJAX. With an RIA, the ease of deployment and maintenance is similar to that of a Web application, assuming that your clients have a reliable network connection. RIA implementations are well suited to Web-based scenarios where you need visualization beyond that provided by basic HTML. They are likely to have more consistent behavior and require less testing across the range of supported browsers when compared to Web applications that utilize advanced functions and code customizations. RIA implementations are also perfect for streaming-media applications. They are less suited to extremely complex multi-page user interfaces (UIs).

- **Design to use a Web infrastructure utilizing services**. RIA implementations require an infrastructure similar to Web applications. Communication to the business layer of your application is usually through services, which allows reuse of existing Web application infrastructure. Transferring logic to the client should only be considered later in the design process. Only transfer logic for performance optimization and UI responsiveness reasons.
- **Design for running in the browser sandbox**. RIA implementations have higher security by default and therefore may not have access to all devices on a machine, such as cameras and hardware video acceleration. Access to the local file system is limited. Local storage is available, but there is a maximum limit.
- **Determine the complexity of your UI requirements**. Consider the complexity of your UI. RIA implementations work best when using a single screen for all operations. They can be extended to multiple screens, but this requires extra code and screen-flow consideration. Users should be able to easily navigate or pause, and return to the appropriate point in a workflow, without restarting the whole process. For multi-page UIs, use deep-linking methods. Also, manipulate the Uniform Resource Locator (URL), the history list, and the browser's back and forward buttons to avoid confusion as users navigate between screens.
- **Use scenarios to increase application performance or responsiveness**. List and examine the common application scenarios to decide how to intelligently divide and load modules, as well as how to cache or move business logic to the client. To reduce the download and start-up time for the application, intelligently segregate functionality into separate downloadable modules. Initially load only code stubs, which can lazy-load other modules. Consider moving or caching regularly used business layer processes on the client for maximum application performance.
- **Design for scenarios where the plug-in is not installed**. Because RIA implementations require a browser plug-in, a you should design for non-interruptive plug-in installation. Consider whether your clients have access to, have permission to, and will want to install the plug-in. Consider what control you have over the installation process. Plan for the scenario where users cannot install the plug-in, by displaying an informative error message, or by providing an alternative Web UI.

RIA Frame

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

Table 1 RIA frame

Category	Key issues
Business Layer	• Moving business operations to the client for reasons other than improved user experience or application performance • Failing to use profiling to identify expensive business operations that should be moved to the client • Trying to move all business processing to the client • Failing to put business rules on the client into their own separate components to allow easy caching, updating, and replacement • Using less powerful browser-supported languages instead of considering windowless RIA plug-ins written in rich programming languages to provide client side processing
Caching	• Failing to use isolated storage appropriately • Failing to check and request increase of the isolated storage quota • Failing to intelligently divide large client applications into smaller separately downloadable components • Downloading and instantiating the entire application at start-up instead of intelligently and dynamically loading modules
Communication	• Trying to use a synchronous communication model • Using an incorrect strategy to bind to the service interface • Attempting to use sockets over unsupported or blocked ports
Controls	• Adding custom control behavior through sub-classing instead of attaching new behavior to specific instances of controls • Incorrect use of controls for a UI type • Implementing custom controls when not required
Composition	• Incorrectly implementing composition patterns, leading to dependencies that require frequent application redeployment • Using composition when not appropriate for the scenario • Not considering composition, and completely rewriting applications that could be reused with minimal or no changes
Data Access	• Performing data access from the client • Failing to filter data at the server
Exception Management	• Failing to design an exception-management strategy • Failing to trap asynchronous call errors and unhandled exceptions; for example, not using the OnError event handler supported by Microsoft Silverlight to trap exceptions in asynchronous calls
Logging	• Failing to log critical errors • Failing to consider a strategy to transfer logs to the server • Segregating logs by machine instead of by user
Media & Graphics	• Failing to take advantage of adaptive streaming for video delivery • Assuming access to hardware acceleration on client

Category	**Key issues**
Presentation	• Not pixel-snapping UI elements, which results in degraded UI appearance • Failing to handle the forward and back button events • Not considering and designing for deep linking when necessary
Portability	• Failing to consider the cost of testing for each platform and browser combination in a Web application compared to using an RIA interface • Using platform-specific APIs in code rather than portable RIA routines • Using less powerful browser-based languages instead of more powerful portable RIA languages
State Management	• Failing to use isolated storage • Using the server to store frequently changing application state • Failing to synchronize state between the client and server when user configuration must be available on multiple clients
Validation	• Failing to identify trust boundaries and validate data that passes across them • Failing to validate data on both the client and the server • Failing to collate extensive client-side validation code into a separate downloadable module

Business Layer

RIA implementations provide the capability to move business processing to the client. Consider moving logic that improves the user experience or performance of the application as a whole.

Consider the following guidelines when designing the business layer:

- Consider starting with your business logic on the server exposed through services. Use scenario-based profiling to discover and target routines that cause the heaviest server load or have the most impact on UI responsiveness. Consider moving or caching only those routines to the client.
- When locating business logic on the client, consider putting business rules or routines in a separate assembly that the application can load and update independently.
- If your business logic is duplicated on the client and the server—for instance, if you are processing business rules on the client for performance but implement the same rules in the service for integrity—use the same code language on the client and server if your RIA implementation allows it. This will reduce any differences in language implementations and make it easier to be consistent in how rules are processed.
- If your RIA implementation allows creation of an instance without a UI, consider using it intelligently. You can keep your processing code in more structured, powerful, or familiar programming languages (such as C#) instead of using less flexible browser-supported languages.
- For security reasons, do not put highly sensitive unencrypted business logic on the client.

Caching

RIA implementations generally use the normal browser caching mechanism. Caching resources intelligently will improve application performance.

Consider the following guidelines when designing a caching strategy:

- Cache components of your application for improved performance and fewer network round trips. Allow the browser to cache objects that are not likely to change during a session. Utilize specific RIA local storage for information that changes during a session, or which should persist between sessions.
- Use installation, updates, and user scenarios to derive intelligent ways to divide and load application modules.
- Load stubs at start-up and then dynamically load additional functionality in the background. Consider using events to intelligently pre-load modules just before they may be required.
- To avoid unintended exceptions, check that isolated storage is large enough to contain the data you will write to it. Storage space does not increase automatically; you must ask the user to increase it.

Communication

RIA implementations must use the asynchronous call model for services to avoid blocking browser processes. Cross-domain, protocol, and service-efficiency issues should be considered as part of your design. If your RIA implementation allows it, consider using threading for background operations.

Consider the following guidelines when designing a communication strategy:

- If you have long-running code, consider using a background thread or asynchronous execution to avoid blocking the UI thread.
- If you are authenticating through services, design your services to use a binding that your RIA implementation supports.
- Ensure that the RIA and the services it calls use compatible bindings that include security information.
- If your RIA client must access a server other than the one from which it was downloaded, ensure that you use a cross-domain configuration mechanism to permit access to the other servers/domains.
- Consider using sockets to push data to the client if client polling causes heavy server load. Consider using sockets to push information to the server when this is significantly more efficient than using services; for example, real-time multi-player gaming scenarios utilizing a central server.

Controls

RIA implementations usually have their own native controls. You can often mix RIA-based and non-RIA-based controls in the same UI, but extra communication code may be required.

Consider the following guidelines when designing a strategy for controls:

- Use native RIA controls where possible.
- If the appropriate control is not supplied with your RIA package, consider third-party RIA-specific controls.

- If a native RIA control is not available, consider using a windowless RIA control in combination with an HTML or Windows Forms control that does have the necessary functionality.
- If your RIA controls support the ability to attach added behaviors, use that ability and avoid sub-classing the controls to extend functionality.

Composition

Composition allows you to implement highly dynamic UIs that you can maintain without changes to the code or redeployment of the application. You can compose an application using RIA and non-RIA components.

Consider the following guidelines when designing a composition strategy:

- Evaluate which composition model patterns best suit your scenario.
- If an interface must gather information from many disparate sources, and those sources are user-configurable or change frequently, consider using composition.
- When migrating an existing HTML application, consider mixing RIA and the existing HTML on the same page to reduce application reengineering.

Data Access

RIA implementations access data in a similar way to normal Web applications. They should request data from the Web server through services in the same way as an AJAX client. After data reaches the client, it can be cached to maximize performance.

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

- Use client-side processing to minimize the number of round trips to the server, and to provide a more responsive UI.
- Filter data at the server rather than at the client to reduce the amount of data that must be sent over the network.
- For operation-based applications, utilize services to access data.

Exception Management

A good exception-management strategy is essential for correctly handling and recovering from errors in any application. In an RIA, you must consider asynchronous exceptions as well as exception coordination between the client and server code.

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

- Do not use exceptions to control business logic.
- Only catch internal exceptions that you can handle. 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. Unhandled exceptions in RIAs are passed to the browser. They will allow execution to continue after the user dismisses a browser error message. Provide a friendly error message for the user if possible. Stop program execution if continued execution would be harmful to the data integrity of the application or could mislead the user into thinking the application is still in a stable state.
- Design an appropriate logging and notification strategy for critical errors and exceptions that does not reveal sensitive information.
- Design for both synchronous and asynchronous exceptions. Use **try/catch** blocks to trap exceptions in synchronous code. Put exception handling for asynchronous service calls in the separate handler designed specifically for such exceptions; for example, in Silverlight, this is the **OnError** handler.

Logging

Logging for the purpose of debugging or auditing can be challenging in an RIA implementation. For example, access to the client file system is not available in Silverlight applications, and execution of the client and the server proceed asynchronously. Log files from a client user must be combined with server log files to gain a full picture of program execution.

Consider the following guidelines when designing a logging strategy:

- Consider the limitations of the logging component in the RIA implementation. Some RIA implementations log each user's information in a separate file, perhaps in different locations on the disk.
- Determine a strategy to transfer client logs to the server for processing. Recombination of different users' logs from the same machine may be necessary if troubleshooting a client machine–specific issue.
- If using isolated storage for logging, consider the maximum size limit and the requirement to ask the user for increases in storage capacity.
- Consider enabling logging and transferring logs to the server when exceptions are encountered.

Media and Graphics

RIA implementations provide a much richer experience and better performance than ordinary Web applications. Research and utilize the built-in media capabilities of your RIA platform. Keep in mind the features that may not be available on the RIA platform compared to a stand-alone media player.

Consider the following guidelines when designing for multimedia and graphics:

- Design to utilize streaming media and video in the browser instead of invoking a separate player utility.
- To increase performance, position media objects on whole pixels and present them in their native size.

- Use adaptive streaming in conjunction with RIA clients to gracefully and seamlessly handle varying bandwidth issues.
- Utilize the native vector graphics engine for the best drawing performance.
- If programming an extremely graphics-intensive application, find out if your RIA implementation provides hardware acceleration. If it does not, create a baseline for what is acceptable drawing performance. Consider a plan to reduce load on the graphics engine if it falls below acceptable limits.

Mobile

RIA implementations provide a much richer experience than an ordinary mobile application. Utilize the built-in media capabilities of the RIA platform you are using.

Consider the following guidelines when designing for mobile device multimedia and graphics:

- When an RIA application needs to be distributed on a mobile client, research whether an RIA plug-in implementation is available for the device you want to support. Find out if the RIA plug-in has reduced functionality compared to non-mobile platforms.
- Attempt to start from a single or similar codebase. Branch code as required for specific devices.
- Re-examine UI layout and implementation for the smaller screen size.
- RIA applications work on mobile devices, but consider using different layout code on each type of device to reduce the impact of different screen sizes when designing for Microsoft Windows Mobile®.

Portability

One of the main benefits of RIAs is the portability of compiled code between different browsers, operating systems, and platforms. Similarly, using a single source codebase, or similar codebases, reduces the time and cost of development and maintenance, while still providing platform flexibility.

Consider the following guidelines when designing for portability:

- Design for the goal of “write once, run everywhere,” but be willing to fork code in cases where overall project complexity or feature tradeoffs dictate that you do so.
- When comparing an RIA and a Web application, consider that differences between browsers can require extensive testing of ASP.NET and JavaScript code. With an RIA application, the plug-in creator, and not the developer, is responsible for consistency across different platforms.
- If your audience will be running the RIA on multiple platforms, do not use features available only on one platform; for example, Windows Integrated Authentication. Design a solution based on standards that are portable across different clients.
- When possible, use richer development languages that are supported for both rich clients and RIAs. See the Technology considerations in this section for recommendations.
- Make full use of the native RIA code libraries.

Presentation

RIA applications work best when designed as one central interface. Multi-page UIs require that you consider how you will link between pages. Positioning of elements on a page can affect both the look and performance of your RIA application.

Consider the following guidelines when designing for presentation:

- To avoid anti-aliasing issues that can cause fuzziness in RIAs, snap UI components to whole pixels. Pay attention to centering and math-based positioning routines. Consider writing a routine that checks for fractional pixels and rounds them to the nearest whole pixel value.
- Trap the browser's forward and back button events to avoid unintentional navigation away from your page.
- For multi-page UIs, use deep-linking methods to allow unique identification of and navigation to individual application pages.
- For multi-page UIs, consider the ability to manipulate the browser's address text box content, history list, and back and forward buttons in order to implement normal Web page–like navigation.

State Management

You can store application state on the client by using isolated storage if the state changes frequently. If application state is vital at start-up, synchronize the client state to the server.

Consider the following guidelines when designing for state management:

- Store state on the client in isolated storage to persist it during and between sessions.
- Store the client state on the server if loss of state on the client would be catastrophic to the application's function. Consider that isolated storage is deleted when the browser cache is cleared.
- Store the client state on the server if the client requires recovery of application state when using different accounts, or when running on other hardware installations
- Verify the stored state between the client and server at start-up, and intelligently handle the case when they are out of synchronization.
- Design for multiple concurrent sessions because you cannot prevent multiple RIA instances from initializing. Design either for concurrency in your state management, or to detect and prevent multiple sessions from corrupting application state.

Validation

Validation must be performed using code on the client or through services located on the server. If you require more than trivial validation on the client, isolate validation logic in a separate downloadable assembly. This makes the rules easy to maintain.

Consider the following guidelines when designing for validation:

- Use client-side validation to maximize the user experience, and server-side validation for security. Use isolated storage to hold client-specific validation rules.

- In general, assume that all client-controlled data is malicious. The server should re-validate all data sent to it. Design to validate input from all sources, such as the query string, cookies, and HTML controls.
- Design to constrain, reject, and sanitize data. Validate input for length, range, format, and type.
- For rules that require access to server resources, evaluate whether it is more efficient to use a single service call that performs validation on the server.
- If you have a large volume of client-side validation code that may change, consider locating it in a separate downloadable module so it can be easily replaced without downloading the entire RIA application again.

Performance Considerations

Properly using the client-side processing power for an RIA is one significant way to maximize performance. Server-side optimizations similar to those used for Web applications are also a major factor.

Consider the following key performance guidelines:

- Cache components of your application for improved performance and fewer network round trips. Allow the browser to cache objects that are not likely to change during a session. Utilize specific RIA local storage for information that changes during a session, or that should be persisted between sessions.
- Use installation, updates, and user scenarios to derive intelligent ways to divide and load application modules. Load stubs at start-up and then dynamically load additional functionality in the background. Consider using events to intelligently pre-load modules just before they may be needed. For example, in a merchant application, wait to load checkout functionality until sometime after a user has added an item to the shopping cart.
- Use scenario-based profiling to discover and target routines that cause the heaviest server load, or that have a major impact on UI responsiveness. Consider moving or caching these routines on the client. When locating business logic on the client, place business rules or routines in a separate assembly that the application can load and update independently.
- Position media objects on whole pixels and present them in their native size. Do not blend them with other objects, such as progress bar controls.
- Be aware of the size of your drawing areas. Only redraw parts of an area that are actually changing. Reduce overlapping regions when not necessary to reduce blending. Use profiling and debugging methods—for example, the “EnableRedrawRegions = true” setting in Silverlight—to see what areas are being redrawn. Note that certain effects, such as blurring, can cause every pixel in an area to be redrawn. Windowless and transparent controls can also cause unintended redrawing and blending.

Security Considerations

RIA applications mitigate a variety of common attack vectors because they run inside a sandbox in the browser. Access to most local resources is limited or restricted, which minimizes opportunities for attacks on the RIA and the client platform on which it runs.

Consider the following restrictions:

- Applications run inside a sandbox in the browser, within a memory space isolated from other applications.
- Browsing of the local client file system is restricted.
- Access to specialized local devices such as Webcams may be limited or not available.
- Access to domains other than the one that delivered the application is limited, protecting the user from cross-site scripting attacks

Protect sensitive information using the follow methods:

- The isolated storage mechanism provides a method for storing data locally, but does not provide built-in security. Do not store sensitive data locally unless it is encrypted using the platform encryption routines.
- Create an exception-management strategy to prevent exposure of sensitive information through unhandled exceptions.
- Be careful when downloading sensitive business logic used on the client because tools are available that can extract the logic contained in downloaded XAML Browser Application (XBAP) files. Implement sensitive business logic using Web services. If the logic must be on the client for performance reasons, research and utilize any available obfuscation methods.
- To minimize the amount of time that sensitive data is available on the client, utilize dynamic loading of resources and overwrite or clear components containing sensitive data from the browser cache.

Deployment Considerations

RIA implementations provide many of the same benefits as Web applications in terms of deployment and maintainability. Design your RIA as separate modules that can be downloaded individually and cached to allow replacement of one module instead of the whole application. Version your application and components so that you can detect the versions that clients are running.

Consider the following guidelines when designing for deployment and maintainability:

- Consider how you will manage the scenario where the RIA browser plug-in is not installed.
- Consider how you will redeploy modules when the application instance is still running on a client.
- Divide the application into logical modules that can be cached separately, and that can be replaced easily without requiring the user to download the entire application again.
- Version your components.

Installation of the RIA plug-in

Consider how you will manage installation of the RIA browser plug-in when it is not already installed:

- **Intranet**. If available, use application distribution software or the Group Policy feature of the Microsoft Active Directory® directory service to pre-install the plug-in on each computer in the organization. Alternatively, consider using Windows Update, where Silverlight is an optional component. Finally, consider manual installation through the browser, which requires the user to have Administrator privileges on the client machine.
- **Internet**. Users must install the plug-in manually, so you should provide a link to the appropriate location to download the latest plug in. For Windows users, Windows Update provides the plug-in as an optional component.
- **Plug-in updates**. In general, updates to the plug-in take into account backward compatibility. You may target a particular plug-in version, but consider implementing a plan to verify your application's functionality on new versions of the browser plug-in as they become available. For intranet scenarios, distribute a new plug-in after testing your application. In Internet scenarios, assume that automatic plug-in updates will occur. Test your application using the plug-in beta to ensure a smooth user transition when the plug-in is released.

Distributed Deployment

Because RIA implementations move presentation logic to the client, a distributed architecture is the most likely scenario for deployment.

In a distributed RIA deployment, the presentation logic is on the client and the business and data layers reside on the Web server or application server. Typically, you will have your business and data access layers on the same server, as shown in Figure 2.

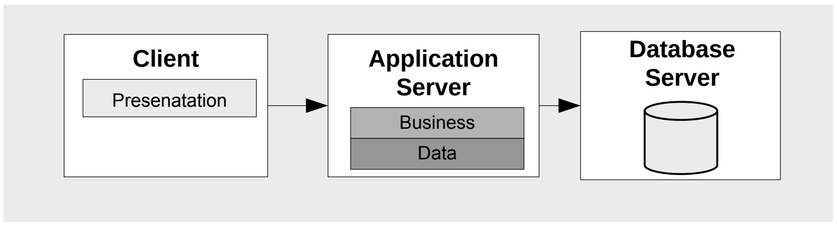

Figure 2 Distributed deployment for an RIA

Consider the following guidelines:

- If your applications are large, factor in the processing requirements for downloading the RIA components to clients.
- If your business logic is shared by other applications, consider using distributed deployment.
- If you use sockets in your application and you are not using port 80, consider which ports you must open in your firewall.
- Ensure that you use a **crossdomain.xml** file so that RIA clients can access other domains where required.

- Design the presentation layer in such as way that it does not initiate, participate in, or vote on atomic transactions.
- Consider using a message-based interface for your business logic.

Load Balancing

When you deploy your application on multiple servers, you can use load balancing to distribute RIA client requests to different servers. This improves response times, increases resource utilization, and maximizes throughput. Figure 3 shows a load-balanced scenario.

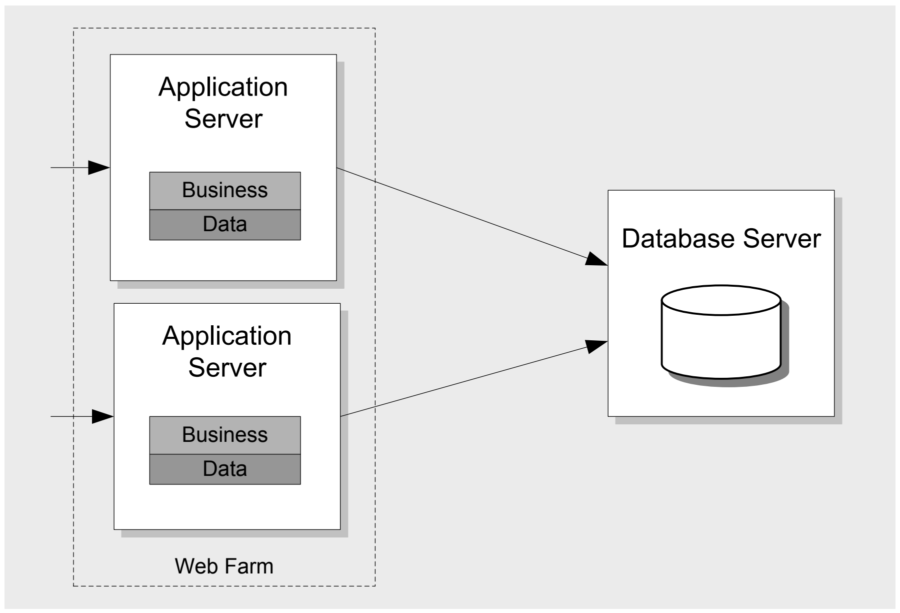

Figure 3 Load balancing an RIA deployment

Consider the following guidelines when designing your application to use load balancing:

- Avoid server affinity. *Server affinity* occurs when all requests from a particular client must be handled by the same server. It is most often introduced by using locally updatable caches or in-process or local session state stores.
- Consider storing all state on the client and designing stateless business components.
- Consider using network load balancing software to implement redirection of requests to the servers in an application farm.

Web Farm Considerations

Consider using a Web farm that distributes requests from RIA clients to multiple servers. A Web farm allows you to scale out your application, and reduces the impact of hardware failures. You can use either load balancing or clustering solutions to add more servers for your application.

Consider the following guidelines:

- Consider using clustering to reduce the impact of hardware failures.
- Consider partitioning your database across multiple database servers if your application has high I/O requirements.

- If you must support server affinity, configure the Web farm to route all requests for the same user to the same server.
- Do not use in-process session management in a Web farm unless you implement server affinity, because requests from the same user cannot be guaranteed to be routed to the same server otherwise. Use the out-of-process session service or a database server for this scenario.

Pattern Map

Table 2 Pattern map

Category	Relevant patterns
Business Layer	• Service Layer
Caching	• Page Cache
Communication	• Asynchronous Callback • Command
Controls	• Chain of Responsibility
Composition	• Composite View • Inversion of Control
Presentation	• Application Controller • Model-View-Controller

Pattern Descriptions

- **Application Controller**. An object that contains all of the flow logic and is used by other Controllers that work with a Model and display the appropriate View.
- **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.
- **Chain of Responsibility**. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- **Command**. Encapsulate request processing in a separate command object that exposes a common execution interface.
- **Composite View.** Combine individual views into a composite view.
- **Inversion of Control**. Populate any dependencies of objects on other objects or components that must be fulfilled before the object can be used by the client application.
- **Model-View-Controller**. Separate the UI code into three separate units; Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Page Cache**. Improve the response time for dynamic Web pages that are accessed frequently, but that change less often and consume a large amount of system resources to construct.
- **Service Layer**. An architectural design pattern where the service interface and implementation is grouped into a single layer.

Technology Considerations

The following guidelines discuss Silverlight and Microsoft Windows Communication Foundation (WCF) and provide specific guidance for these technologies. At the time of writing, the latest versions are WCF 3.5 and Silverlight 2.0. Use the guidelines to help you to choose and implement an appropriate technology:

- At the time of this guide's release, Silverlight for Mobile was an announced product and in development, but not released.
- Silverlight currently supports the Safari, Firefox, and Microsoft Internet Explorer browsers though a plug-in. Through these browsers, Silverlight 2.0 currently supports Mac, Linux, and Microsoft Windows®. Support for Windows Mobile was also announced in 2008.
- The local storage mechanism for Silverlight is named "Isolated Storage." The current initial size is 1 megabyte (MB). The maximum storage size is unlimited. Silverlight requires that you ask the user to increase the storage size.
- Silverlight only supports Basic HTTP binding. WCF in .NET 3.5 supports Basic HTTP binding, but security is not turned on by default. Be sure to turn on at least transport security to secure your service communications.
- Silverlight does not obfuscate modules downloaded as XBAPs. XBAPs can be decompiled and the programming logic extracted.
- The .NET cryptography APIs are available in Silverlight and should be utilized when storing and communicating sensitive data to the server if not already encrypted using another mechanism.
- Silverlight contains controls specifically designed for it. Third parties are likely to have additional control packages available.
- Use Silverlight windowless controls if you want to overlay viewable HTML content and controls on top of a Silverlight application.
- If you want to provide rich control functionality in cases when WPF is not available (such as when Microsoft .NET Framework 3.5 is not installed on the machine), you can embed a Silverlight application in Windows Forms and run it in the Web Browser control.
- Silverlight allows you to attach additional behaviors to existing control implementations. Use this approach instead of attempting to subclass a control.
- Silverlight supports only asynchronous calls to Web services.
- Silverlight calls use the **OnError** event handler for an application when exceptions occur in services, or when synchronous exceptions are not handled.
- Silverlight does not currently support SOAP faults exposed by services due to the browser security model. Services must return exceptions to the client through a different mechanism.
- Silverlight supports two file formats to deal with calling services cross-domain. You can use either a ClientAccessPolicy.xml file specific to Silverlight, or a CrossDomain.xml file compatible with Adobe Flash. Place the file in the root of the server(s) to which your Silverlight client needs access.
- In Silverlight, you must implement custom code for input and data validation. Check documentation to verify whether this is true for later versions.

- Silverlight performs anti-aliasing for all UI components, so consider the recommendations in the Presentation section about snapping UI elements to whole pixels.
- Consider using ADO.NET Data Services in a Silverlight application if large amounts of data must be transferred from the server.
- Silverlight logs to an individual file in the user store for a specific logged-in user. It cannot log to one file for the whole machine.
- Silverlight supports the languages of C#, Iron Python, Iron Ruby, and VB.NET. Most XAML code will also run in both WPF and Silverlight hosts.

Additional Resources

- For official information on Silverlight, see the official Silverlight Web site at http://silverlight.net/default.aspx
- For Silverlight blogs, see http://blogs.msdn.com/brada/ and http://weblogs.asp.net/Scottgu/

Chapter 17: Rich Client Applications

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 for rich client applications.
- Understand deployment scenarios for rich client applications.
- Learn the key patterns and technology considerations for designing rich client applications.

Overview

Rich client user interfaces (UIs) 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, Windows Presentation Foundation (WPF), and Microsoft® 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 stand-alone 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 typical rich client architecture, and identifies the components usually found in each layer.

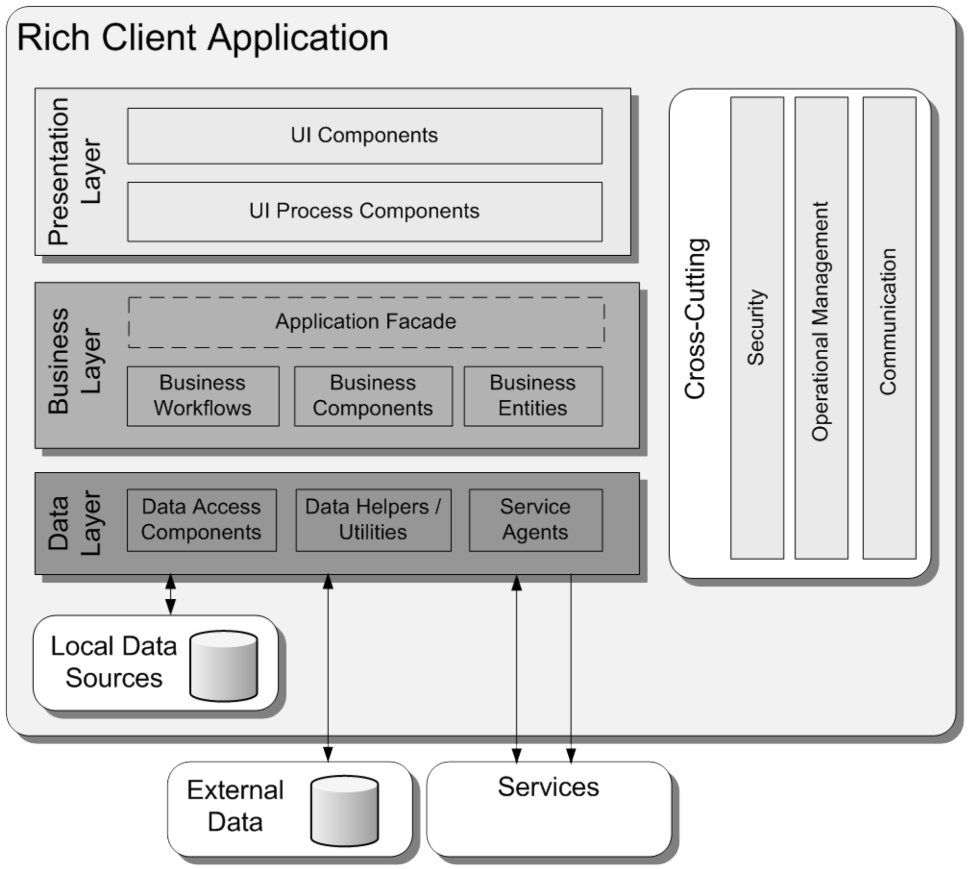

Figure 1 Overall view of typical rich client architecture

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 only communicate with other layers to consume or send back information. An example is the difference between an application such as a File Transfer Protocol (FTP) client that depends on a remote server to provide all of the data for display, and an application such as Microsoft Office 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 terms of the presentation layer and UI, 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 applications, and rely only on data retrieved and sent back to other layers of the application and to other applications. These types of rich clients 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 UI elements themselves, and the same basic user interaction patterns can be reused by multiple UIs.
- **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 façade** (optional). A façade is used to combine multiple business operations into single message-based operation. You might access the application façade from the presentation layer by 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 might 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 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.

Consider the following guidelines when designing rich client applications:

- **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 Model-View-Controller (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 any 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 UI.

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.

Table 1 Common rich client design issues

Category	Key issues
Communication	• Choosing the wrong communication protocols and technology • Using synchronous communication methods when asynchronous methods could provide better responsiveness • 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 Access	• 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 • Controlling application flow with exceptions
State Management	• Failing to store and manage UI state information correctly • Failing to cache state where 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 by using a variety of protocols and methods. These may include Hypertext Transfer Protocol (HTTP) requests, Simple Mail Transfer Protocol (SMTP) e-mail messaging, SOAP Web service messages, Simple Network Time Protocol (SNTP) time synchronization packets, Distributed Component Object Model (DCOM) for remote components, and many other Transmission Control Protocol (TCP)/Internet Protocol (IP)–based standard or custom communication protocols. Alternatively, if the client application is located on the same physical tier as the business layer, you should use object-based interfaces to interact with the business layer.

Consider the following guidelines when designing a communication strategy:

- 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 Internet Protocol Security (IPSec) and Secure Sockets Layer (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 potential performance and network impact. Choose more efficient communication protocols such as 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 need to support open communication standards.

Composition

Complex UIs 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 run time. 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.

Consider the following guidelines when designing a composition strategy:

- 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 startup, 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.

Consider the following guidelines when designing a configuration management strategy:

- Determine what configurable data may change in the life of your application; for example, file locations, developer versus 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 might affect or override local configurations.

Data Access

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.

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

- If the client application cannot handle the data in the exposed format, you must implement a translation mechanism that converts it. However, 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 UI is still responsive while the data is loading. However, you must also be aware of conflicts that might 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 in order 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 consider logging errors and exceptions for use by management tools and monitoring systems.

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

- Identify the errors and exceptions that are 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 optionally 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 and logging unhandled exceptions. 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 in order to prevent sensitive information from 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.

Consider the following guidelines when designing a state-management strategy:
- 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 re-creating or re-fetching the data. These factors will help you decide 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.
- Consider at what granularity you need to maintain state information. For instance, determine the state information that applies to all users of an application versus the information that applies only to specific users or roles

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 or custom solutions, or you can take advantage of a framework such as Windows Workflow (WF) or, for document-based interfaces, Microsoft Office SharePoint® Server (MOSS).

Consider the following guidelines when designing a workflow strategy:
- 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 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 partially 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. Presentation layer is the parts 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 UI can have a negative impact on a business application that performs well in all other respects.

Consider the following guidelines when designing the presentation features of your application:

- 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 might 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 Layer Considerations

In most business scenarios, rich client applications will 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 might consider locating some of the business-processing tasks on the client.

Consider the following guidelines when designing interaction with business and service layers:

- 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 might want 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. Issues to consider include deployment, updates, patches, and versioning.

Consider the following guidelines when designing a maintainability strategy:
- 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 in which your application will be used. For example, you might need an installation program for applications that are publically available, or you might 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, run-time 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 run-time 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.

Consider the following guidelines when designing a security strategy:

- 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 might need to create custom validation mechanisms, or you might be able to take advantage of validation frameworks. The Microsoft 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 might 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 use by rich client applications falls into two categories:

- **Read-only reference data**. This is 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 in order 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**. This is 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 that 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 it was last retrieved.

Using ADO.NET DataSets to Manage Data
The ADO.NET **DataSet** helps clients to work with data while offline. **DataSets** 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. **DataSets** can also be used to merge data from different sources.

Windows Forms Data Binding
Windows Forms data binding supports bidirectional binding that allows you to bind a data structure to a UI 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 UI.
- Provide master-detail views of data.
- Allow users to explore complex related data items.
- Provide lookup table functionality, allowing the UI 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 reestablished, 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 might 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.

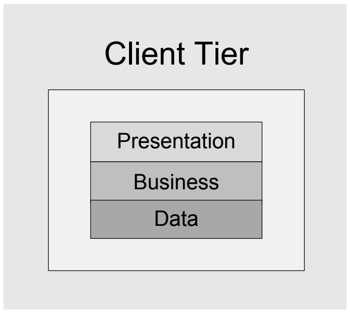

Figure 2 Stand-alone deployment for a rich client application

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.

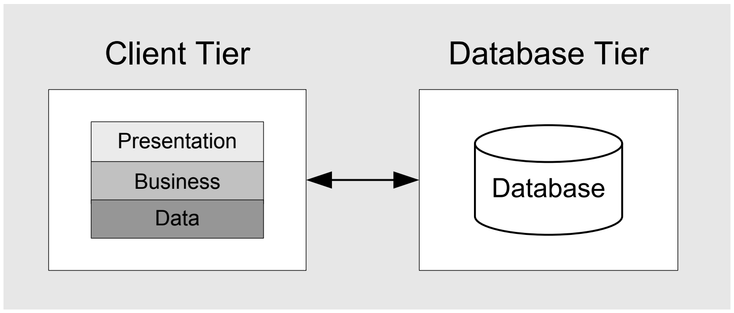

Figure 3 Client/server deployment for a rich client application

N-Tier

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

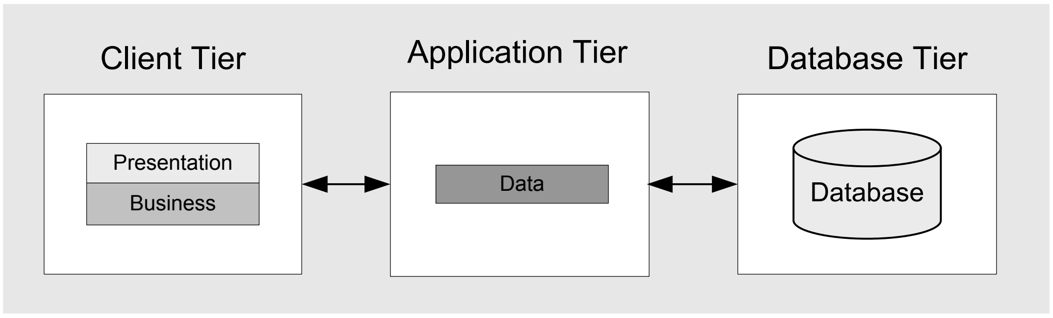

Figure 4 ***N*-tier deployment with the business layer located on the client tier**

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

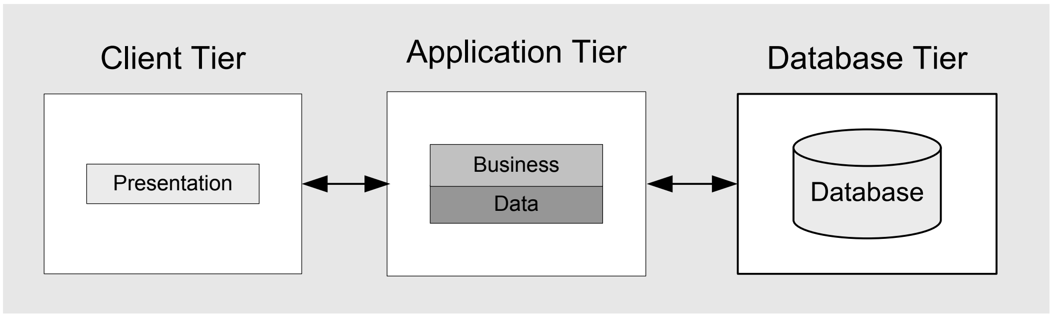

Figure 5 ***N*-tier deployment with the business layer located on the application tier**

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**. This is 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 designing a rich client application:

- Know your target physical deployment environment early, from the planning stage of the life cycle.
- 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

Table 2 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 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.
- **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 in order 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 predefined 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 and interactivity, and that 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 XBAP.
- If you want to build applications that are predominantly document-based, or are used for reporting, consider designing an OBA.
- 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 if you need to obtain Microsoft 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 Windows Management Instrumentation (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 18: Services

Objectives

- Understand the nature and use of services.
- Learn the general guidelines for different service scenarios.
- Learn the guidelines for the key attributes of services.
- Learn the guidelines for the layers within a services application.
- Learn the guidelines for performance, security, and deployment.
- Learn the key patterns and technology considerations related to services.

Overview

A *service* is a public interface that provides access to a unit of functionality. Services literally provide some programmatic “service” to the caller, who consumes the service. Services are loosely coupled and can be combined within a client or within other services to provide more complex functionality. Services are distributable and can be accessed from a remote machine as well as from the local machine on which they are running. Services are message-oriented, meaning that service interfaces are defined by a Web Services Description Language (WSDL) file, and operations are called using Extensible Markup Language (XML)–based message schemas that are passed over a transport channel. Services support a heterogeneous environment by focusing interoperability at the message/interface definition. If components can understand the message and interface definition, they can use the service regardless of their base technology.

Figure 1 shows an overall view of common services application architecture.

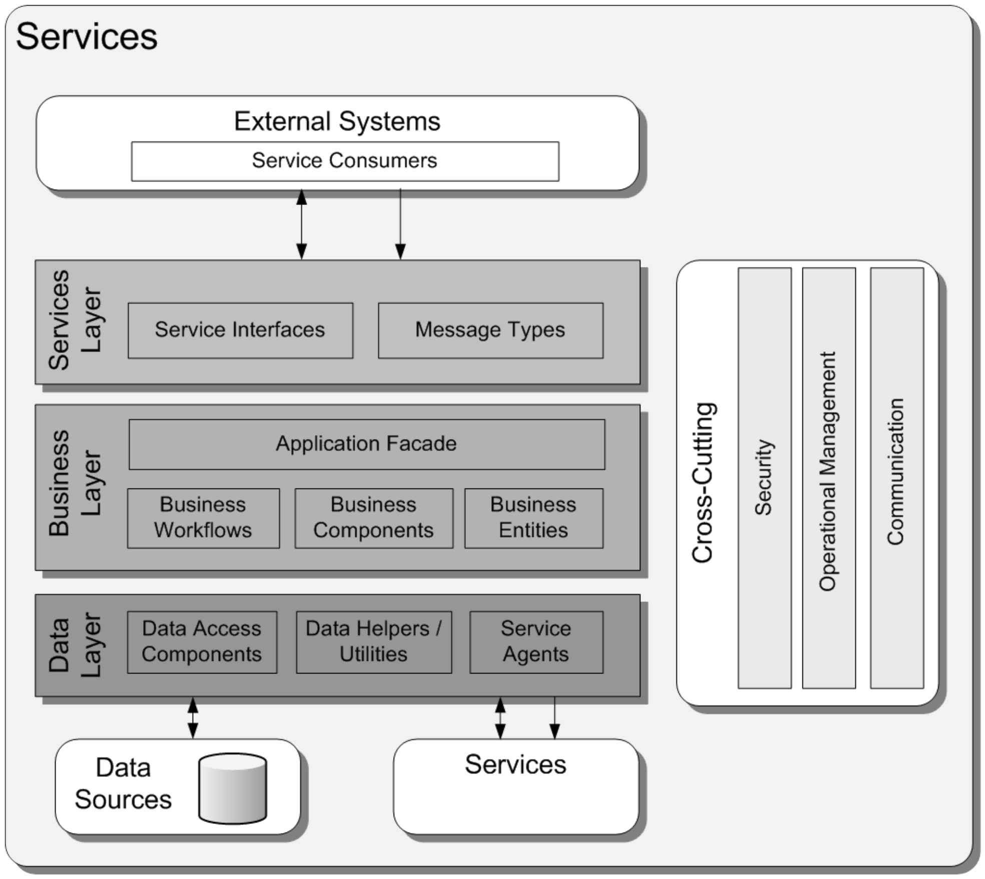

Figure 1 Common services application architecture

Common Services Scenarios

Services are flexible by nature and can be used in a wide variety of scenarios and combinations. The following are key typical scenarios:

- **Service exposed over the Internet**. This scenario describes a service that is consumed over the Internet. Decisions on authentication and authorization must be based on Internet trust boundaries and credentials options. For example, username authentication is more likely in the Internet scenario than the intranet scenario. This scenario includes business-to-business (B2B) as well as consumer-focused services. A Web site that allows you to schedule visits to your family doctor would be an example of this scenario.
- **Service exposed over an intranet**. This scenario describes a service that is consumed over an intranet. Decisions on authentication and authorization must be based on intranet trust boundaries and credentials options. For example, the Microsoft® Active Directory® directory service is more likely to be the chosen user store in the intranet scenario than in the Internet scenario. An enterprise Web-mail application would be an example of this scenario.
- **Service exposed on the local machine**. This scenario describes a service that is consumed by an application on the local machine. Transport and message-protection decisions must be based on local machine trust boundaries and users.
- **Mixed scenario**. This scenario describes a service that is consumed by multiple applications over the Internet, an intranet, and/or the local machine. A line-of-business (LOB) application

that is consumed internally by a thick client application and over the Internet by a Web application would be an example of this scenario.

Design Considerations

When designing service-based applications, you should follow the general guidelines that apply to all services, such as designing for coarse-grained operations, honoring the service contract, and anticipating invalid requests or invalid request orders. In addition to the general guidelines, there are specific guidelines that you should follow for different types of services. For example, with a Service-Oriented Architecture (SOA), you should ensure that the operations are application-scoped and that the service is autonomous. Alternatively, you might have an application that provides workflow services, or you might be designing an operational data store (ODS) that provides a service-based interface.

General Considerations

- **Design coarse-grained operations**. Avoid chatty calls to the service interface, which can lead to very poor performance. Instead, use the Façade pattern to package smaller fine-grained operations into single coarse-grained operations.
- **Design entities for extensibility**. Data contracts should be designed so that you can extend them without affecting consumers of the service.
- **Compose entities from standard elements**. When possible, compose the complex types used by your service from standard elements.
- **Design without the assumption that you know who the client is**. You should not make assumptions about the client, and how they plan to use the service you provide.
- **Design only for the service contract**. Do 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**. 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, or take advantage of infrastructure services, 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 might arrive out of order, implement a design that will store messages and then process them in the correct order.

SOA Services Considerations

- **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.

- **Decouple the interface from the implementation**. In an SOA application, it is very important to keep internal business entities hidden from external clients. In other words, you should never define a service interface that exposes internal business entities. Instead, the service interface should be based on a contract with which external consumers interact. Inside the service implementation, you translate between internal business entities and external contracts.
- **Design services with explicit boundaries**. A services application should be self-contained, with strict boundaries. Access to the service should only be allowed through the service interface layer.
- **Design services to be autonomous**. Services should not require anything from consumers of the service, and should not assume who the consumer is. In addition, you should design services with the assumption that malformed requests will be sent to it.
- **Design compatibility based on policy**. The service should publish a policy that describes how consumers can interact with the service. This is more important for public services, where consumers can examine the policy to determine interaction requirements.

Data Services Considerations

- **Avoid using services to expose individual tables in a database**. This will lead to chatty service calls and interdependencies between service operations, which can lead to dependency issues for consumers of the service.
- **Do not implement business rules with data services**. Different consumers of the data will have unique viewpoints and rules. Attempting to implement rules in data access services will impose restrictions on the use of that data.

Workflow Services Considerations

- **Use interfaces supported by your workflow engine.** Attempting to create custom interfaces can restrict the types of operations supported, and will increase the effort required to extend and maintain the services.
- **Design a service that is dedicated to supporting workflow**. Instead of adding workflow services to an existing service application, consider designing an autonomous service that supports only workflow requirements.

Services Application Frame

The following table lists the key areas to consider as you develop services 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

Area	Key issues
Communication	• Incorrect choice of transport protocol • Chatty communication with the service • Failing to protect sensitive data
Data Consistency	• Failing to check data for consistency • Improper handling of transactions in a disconnected model
Exception Management	• Using exceptions to control application flow • Not logging exceptions • Compromising message integrity when an exception occurs • Revealing sensitive information in the exception • Failing to implement a strategy for unhandled exceptions
Message Construction	• Not appreciating that message contents may be time-sensitive • Incorrect message construction for the operation • Passing too much data in a single message
Message Endpoint	• Not supporting idempotent operations • Implementing filters to handle specific messages • Subscribing to an endpoint while disconnected
Message Protection	• Not protecting sensitive data • Using transport layer protection for messages that cross multiple servers • Not considering data integrity
Message Transformation	• Unnecessary use of transformations • Implementing transformations at the wrong location • Using a canonical model when not necessary
Message Exchange Patterns	• Using complex patterns when not necessary • Using the Request/Response pattern for one-way messages
REST	• Overuse of POST operations against resources • Putting actions into the URI with **QueryString** values • Using session state within a Representational State Transfer (REST) service
SOAP	• Not choosing the appropriate security model • Not planning for fault conditions • Using complex types in the message schema
Validation	• Not validating message structures sent to the service • Failing to validate data fields associated with the message

Authentication

The design of an effective authentication strategy for your service depends on the type of service host you are using. For example, if the service is hosted in Internet Information Services (IIS), you can take advantage of the authentication support provided by IIS. If the service is hosted by using a Windows Service or a console application, you must use message-based authentication.

Consider the following guidelines when designing an authentication strategy:

- Identify a suitable mechanism for securely authenticating users.

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

Authorization

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

Consider the following guidelines when designing an authorization strategy:

- Set appropriate access permissions on resources for users, groups, and roles.
- Consider using Uniform Resource Locator (URL) authorization and/or file authorization when protecting URL- and file-based resources.
- Where appropriate, restrict access to publicly accessible service methods using declarative principle permission demands.
- Execute services under the most restrictive account that is appropriate.

Data Consistency

Designing for data consistency is critical to the stability and integrity of your service implementation. Failure 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.

Consider the following guidelines when designing for data consistency:

- 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.

Communication

When designing the communication strategy for your service application, the protocol you choose should be based on the deployment scenario your service must support. If the service will be deployed within a closed network, you can use Transmission Control Protocol (TCP) for more efficient communication. If the service will be deployed in to a public-facing network, you should choose the Hypertext Transfer Protocol (HTTP).

Consider the following guidelines when designing a communication strategy:

- Determine how to reliably handle unreliable or intermittent communication.
- Use dynamic URL behavior with configured endpoints for maximum flexibility.
- Validate endpoint addresses in messages.
- Determine how to handle asynchronous calls.

- Decide if message communication must be one-way or two-way.

Exception Management

Designing an effective exception-management strategy is important for the security and reliability of your service application. Failure to do so can leave 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, so it is important that the design take into account the potential impact on performance. A good approach is to design a centralized exception-management and logging mechanism, and to consider providing access points that support instrumentation and centralized monitoring in order to assist system administrators.

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

Message Construction

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

Consider the following guidelines when designing a message-construction strategy:
- Determine the appropriate patterns for message constructions (such as Command, Document, Event, and Request-Reply).
- Divide very large quantities of data into relatively small 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 provides the message endpoint. When designing the service implementation, you must consider the type of message that you are consuming. In addition, you should design for a range of scenarios related to handling messages.

Consider the following guidelines when designing message endpoints:

- Determine relevant patterns for message endpoints (such as Gateway, Mapper, Competing Consumers, and Message Dispatcher).
- Determine if you should accept all messages, or implement a filter to handle specific messages.
- Design for idempotency in your service 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 that all duplicate messages will be ignored.
- Design for commutativity. *Commutativity* is the situation where the messages could arrive out of order. In other words, a commutative endpoint will guarantee that messages arriving out of order will be stored and then processed in the correct order.
- Design for disconnected scenarios. For instance, you might need to support guaranteed delivery.

Message Protection

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

Consider the following guidelines when designing message protection:

- If your messages will not be routed intermediary servers, consider using transport layer security such as SSL.
- If the message passes through one or more intermediary servers, use message-based protection, since with transport security alone the message will be decrypted and re-encrypted at each server it passes through.
- Use encryption to protect sensitive data in messages.
- Consider using digital signatures to protect messages and parameters from tampering.

Message Transformation

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

Consider the following guidelines when designing for message transformation:

- Determine relevant patterns for message transformation. For example, the Normalizer pattern can be used to translate semantically equivalent messages into a common format.
- Use metadata to define the message format.

- Consider using an external repository to store the metadata.

Message Exchange Patterns

A Message Exchange Pattern (MEP) defines a conversation between a service and the service consumer. This conversation represents a contract between the service and the consumer. The W3C standards group defines two patterns for SOAP messages: Request-Response and SOAP Response. Another standards group named OASIS has defined a Business Process Execution Language (BPEL) for services. BPEL defines a process for exchanging business process messages. In addition, other organizations have defined specific message-exchange patterns for exchanging business process messages.

Consider the following guidelines when designing message exchange patterns:

- Choose patterns that match your requirements without adding unnecessary complexity. For example, avoid using a complex business process exchange pattern if the Request/Response pattern is sufficient.
- When using business process modeling techniques, be careful not to design exchange patterns based on process steps. Instead, the patterns should support operations that combine process steps.
- Use existing standards for message exchange patterns instead of inventing your own. This promotes a standards-based interface that will be understood by many consumers. In other words, consumers will be more inclined to interact with standards-based contracts instead of having to discover and adapt to nonstandard contracts.

Representational State Transfer (REST)

Representational State Transfer (REST) is an architecture style that is based on HTTP and works very much like a Web application. However, instead of a user interacting with and navigating through Web pages, applications interact with and navigate through REST resources using the same semantics as a Web application. In REST, a resource is identified by a Uniform Resource Identifier (URI), and the actions that can be performed against a resource are defined by using HTTP verbs such as GET, POST, PUT, and DELETE. Interaction with a REST service is accomplished by performing HTTP operations against a URI, which is typically in the form of an HTTP-based URL. The result of an operation provides a representation of the current state for that resource. In addition, the result can contain links to other resources that you can move to from the current resource.

The most common misconception about REST is that it is only useful for Create, Read, Update, and Delete (CRUD) operations against a resource. However, REST can be used with any service that can be represented as a state machine. In other words, as long as you can break a service down into distinguishable states, such as “retrieved” and “updated,” you can convert those states into actions and demonstrate how each state can lead to one or more states. Consider how the user interface (UI) of a Web application represents a state machine. When you access a page, the information displayed represents the current state of that information. You might have the ability to change that state by POSTing form fields, or by moving to another page using links that are included in the current page. A RESTful service works the same way, in that an application can perform a GET operation on a resource to get the current state of that resource, change the state of the resource by performing a PUT operation, or move to a different resource using links provided by the current resource.

Both application state and resource state exist in RESTful services. The client stores all application state, while the server stores only the resource state. Each individual request sent from the client to the server must contain all of the information necessary for the server to fully understand that request. As such, the client must transfer any relevant application state in its request. The client can then make decisions on how to modify the resource state dependent on the server responses. Passing the application state each time allows the application design to scale, as you can now add multiple identical Web servers and load-balance in such a way that the client needs no affinity to one particular server or any shared application state.

A REST style service has the qualities of safety and idempotence. Safety refers to the ability to repeat a request many times and get back the same answer without side effects. Idempotent refers to behavior where making a single call has the same consequences as making the same call a series of times. The presence of these qualities adds robustness and reliability because, even if HTTP is unreliable, you can safely reissue a request when the server is non-responsive or returns a failure.

Consider the following guidelines when designing REST resources:

- Consider using a state diagram to model and define resources that will be supported by your REST service.
- Choose an approach for resource identification. A good practice would be to use meaningful names for REST starting points and unique identifiers, as part of their overall path, for specific resource instances.
- Decide if multiple representations should be supported for different resources. For example, decide if the resource should support an XML, Atom, or JSON format and make it part of the resource request. A resource could be exposed as both http://www.microsoft.com/example.atom and http://www.microsoft.com/example.json .
- Do not use QueryString variables to define actions on a URI. Instead, all actions are based on the HTTP operation performed against a URI.
- Do not overuse the POST operation. A good practice is to use specific HTTP operations such as PUT or DELETE as appropriate to reinforce the resource-based design and the use of a uniform interface.
- Take advantage of the HTTP application protocol to use common Web infrastructure (caching, ETags, authentication, and common data representation types, etc.).
- Ensure that your GET requests are safe, meaning that they always return the same result when called. Consider making your PUT and DELETE requests idempotent, meaning that repeated identical requests should have the same effect as a single request.

SOAP

SOAP is a message-based protocol in which the message is composed of an XML envelope that contains a header and body. The header can provide information that is external to the operation performed by the service. For example, a header may contain security, transaction, or routing information. The body contains contracts, in the form of XML schemas, which define the service and the actions it can perform.

Compared to REST, SOAP gives more protocol flexibility, so you can utilize higher-performance protocols such as Transmission Control Protocol (TCP). SOAP supports WS-* standards including security, transactions, and reliability. Message security and reliability ensure that the messages not only reach their destination, but also that those messages have not been read or modified during transit. Transactions provide the ability to group operations and provide rollback ability in the case of a failure.

SOAP is useful when performing RPC-type interactions between services or decoupled layers of an application. It excels at providing an interface between new and legacy systems on an internal network. A service layer can be placed on top of an older system, allowing API-type interaction with the system without having to redesign the system to expose a REST resource model. SOAP is also useful where information is actively routed to one or more systems that may change communication protocols frequently over their lifetimes. It is also helpful when you want to encapsulate information or objects in an opaque manner and then store or relay that information to another system.

If you want your application to scale through the use of web farms or load balancing, avoid storing session state on the server. Storing sessions on the server means that a particular server must service the client through the duration of the session or must pass the session information to another server in the case of load balancing. Passing session state between servers makes failover and scale-out scenarios much harder to implement.

Consider the following guidelines when designing SOAP messages:
- Within a SOAP envelope, the SOAP header is optional, while the SOAP body is mandatory.
- Consider using SOAP faults for errors instead of relying on default error-handling.
- When returning error information, the SOAP fault can be the only child element within a SOAP body.
- Errors that occur when processing the SOAP header should be returned as a SOAP fault in the SOAP header element.
- In order to force processing of a SOAP header block, set the block’s **mustUnderstand** attribute to “true” or “1”.
- Research and utilize WS-* standards. These standards provide consistent rules and methods for dealing with the issues commonly encountered in a messaging architecture.

Validation

Designing an effective input-validation and data-validation strategy is critical to the security of your application. Determine the validation rules for data you receive from consumers of the service. Understand your trust boundaries so that you can validate any data that crosses these boundaries.

Consider the following guidelines when designing a validation strategy:

- Validate all data received by the service interface.
- Consider the way that data will be used. For example, if the data will be used in database queries, you must protect the database from SQL injection attacks.
- Understand your trust boundaries so that you can validate data that crosses these boundaries.
- Determine if validation that occurs in other layers is sufficient. If data is already trusted, you might not need to validate it again.
- Return informative error messages if validation fails.

Service Layer Considerations

The service layer contains components that define and implement services for your application. Specifically, this layer contains the service interface (which is composed of contracts), the service implementation, and translators that convert internal business entities to and from external data contracts.

Consider the following guidelines when designing your service layer:

- Do not implement business rules in the service layer. The service layer should only be responsible for managing service requests and for translating data contracts into entities for use by the business layer.
- Access to the service layer should be defined by policies. Policies provide a way for consumers of the service to determine the connection and security requirements, as well as other details related to interacting with the service.
- Use separate assemblies for major components in the service layer. For example, the interface, implementation, data contracts, service contracts, fault contracts, and translators should all be separated into their own assemblies.
- Interaction with the business layer should only be through a well-known public interface. The service layer should never call the underlying business logic components.
- The service layer should have knowledge of business entities used by the business layer. This is typically achieved by creating a separate assembly that contains business entities that are shared between the service layer and business layer.

Business Layer Considerations

The business layer in a services application uses a façade to translate service operations into business operations. The primary goal when designing a service interface is to use coarse-grained operations, which can internally translate into multiple business operations. The business layer façade is responsible for interacting with the appropriate business process components.

Consider the following guidelines when designing you business layer:

- Components in the business layer should have no knowledge of the service layer. The business layer and any business logic code should not have dependencies on code in the service layer, and should never execute code in the service layer.
- When supporting services, use a façade in the business layer. The façade represents the main entry point into the business layer. Its responsibility is to accept coarse-grained operations and break them down into multiple business operations.
- Design the business layer to be stateless. Service operations should contain all of the information, including state information, which the business layer uses to process a request. Because a service can handle a large number of consumer interactions, attempting to maintain state in the business layer would consume considerable resources in order to maintain state for each unique consumer. This would restrict the number of requests that a service could handle at any one time.
- Implement all business entities within a separate assembly. This assembly represents a shared component that can be used by both the business layer and the data access layer.

Data Layer Considerations

The data layer in a services application includes the data access functionality that interacts with external data sources. These data sources could be databases, other services, the file system, Microsoft Office SharePoint® lists, or any other applications that manage data.

Consider the following guidelines when designing your data layer:

- The data layer should be deployed to the same tier as the business layer. Deploying the data layer on a separate physical tier will require serialization of objects as they cross physical boundaries.
- Always use abstraction when providing an interface to the data access layer. This is normally achieved by using the Data Access or Table Data Gateway pattern, which use well-known types for inputs and outputs.
- For simple CRUD operations, consider creating a class for each table or view in the database. This represents the Table Module pattern, where each table has a corresponding class with operations that interact with the table.
- Avoid using impersonation or delegation to access the data layer. Instead, use a common entity to access the data access layer, while providing user identity information so that log and audit processes can associate users with the actions they perform.

Performance Considerations

When designing the service interface, use operations that are as coarse-grained as possible. The level of granularity will depend on the type of service you are designing. For example, you would define an operation that returns all of the demographic data for a person in one call, rather than returning portions of the demographic data across multiple calls. Think of a service as an application that provides operations that are used to support business processes. Avoid creating a web of dependencies between services and service consumers, and the chatty communications that would result from a component-based design.

Consider the following guidelines when designing for maximum performance:

- Keep service contract operations as coarse-grained as possible.
- Do not mix business logic with translator logic. The service implementation is responsible for translations, and you should never include business logic in the translation process. The goal is to minimize complexity and improve performance by designing translators that are only responsible for translating data from one format to another.

Security Considerations

Security encompasses a range of factors and is vital in all types of applications. Services applications must be designed and implemented to maximize security and, where they expose business functions, must play their part in protecting and securing the business rules, data, and functionality. 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. However, one specific area of concern for services is protecting messages in transit over the network.

Consider the following guidelines when designing a security strategy:

- When using message-based authentication, you must protect the credentials contained in the message. This can be accomplished by using encryption, or encryption combined with digital signatures.
- You can use transport layer security such as SSL. However, if your service passes through other servers, consider implementing message-based security. This is required because each time a message secured with transport layer security passes through another server, that server decrypts the message and then re-encrypts it before sending it on to the next server.
- When designing extranet or business-to-business (B2B) services, consider using message-based brokered authentication with X.509 certificates. In the B2B scenario, the certificate should be issued by a commercial certificate authority. For extranet services, you can use certificates issued through an organization-based certificate service.

Deployment Considerations

Services applications are usually designed using the layered approach, where the service interface, business, and data layers are decoupled from each other. You can use distributed deployment for a services application in exactly the same way as any other application type. Services may be deployed to a client, a single server, or multiple servers across an enterprise. However, when deploying a services application, you must consider the performance and security issues inherent in distributed scenarios, and take into account any limitations imposed by the production environment.

Consider the following guidelines when deploying a services application:

- Locate the service layer on the same tier as the business layer to improve application performance.
- When a service is located on the same physical tier as the consumer of the service, consider using named pipes or shared memory for communication.
- If the service is accessed only by other applications within a local network, consider using TCP for communication.
- Configure the service host to use transport layer security only if consumers have direct access to the service without passing through other servers or services.

Pattern Map

Category	**Relevant patterns**
Communication	• Duplex • Fire and Forget • Reliable Sessions • Request Response
Data Consistency	• Atomic Transactions • Cross-service Transactions • Long-running transactions
Message Construction	• Command Message • Document Message • Event Message
Message Endpoint	• Competing Consumer • Durable Subscriber • Idempotent Receiver • Message Dispatcher • Messaging Gateway • Messaging Mapper • Polling Consumer • Selective Consumer • Service Activator • Transactional Client
Message Protection	• Data Confidentiality • Data Integrity • Data Origin Authentication • Exception Shielding • Federation • Replay Protection • Validation

Category	Relevant patterns
Message 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 Contract • Fault Contract • Service Contract

Pattern Descriptions

- **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.
- **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.
- **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 applications.

- **Duplex**. Two-way message communication where both the service and the client send messages to each other independently, irrespective of the use of the one-way or 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.
- **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 reduced 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 transactions**. Transactions that are part of a workflow process.
- **Message Dispatcher**. A component that sends messages to multiple consumers.
- **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.
- **Polling Consumer**. A service consumer that checks the channel for messages at regular intervals.
- **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.

- **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 the PUT operation.
- **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 technical considerations should be considered when designing a 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 maximum feature availability and interoperability.
- 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–Microsoft 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 on distributed systems, see *Enterprise Solution Patterns Using Microsoft .NET - Distributed Systems Patterns* at http://msdn.microsoft.com/en-us/library/ms998483.aspx
- For more information on Enterprise Service Bus scenarios, see *Microsoft ESB Guidance for BizTalk Server 2006 R2* at http://msdn.microsoft.com/en-us/library/cc487894.aspx.
- For more information on integration patterns, see *Prescriptive Architecture Integration Patterns* at http://msdn.microsoft.com/en-us/library/ms978729.aspx.

- For more information on service patterns, see *Enterprise Solution Patterns Using Microsoft .NET - Services Patterns* at http://msdn.microsoft.com/en-us/library/ms998508.aspx
- For more information on Web services security patterns, see *Web Service Security* at http://msdn.microsoft.com/en-us/library/aa480545.aspx.

Chapter 19: Mobile Applications

Objectives

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

Overview

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

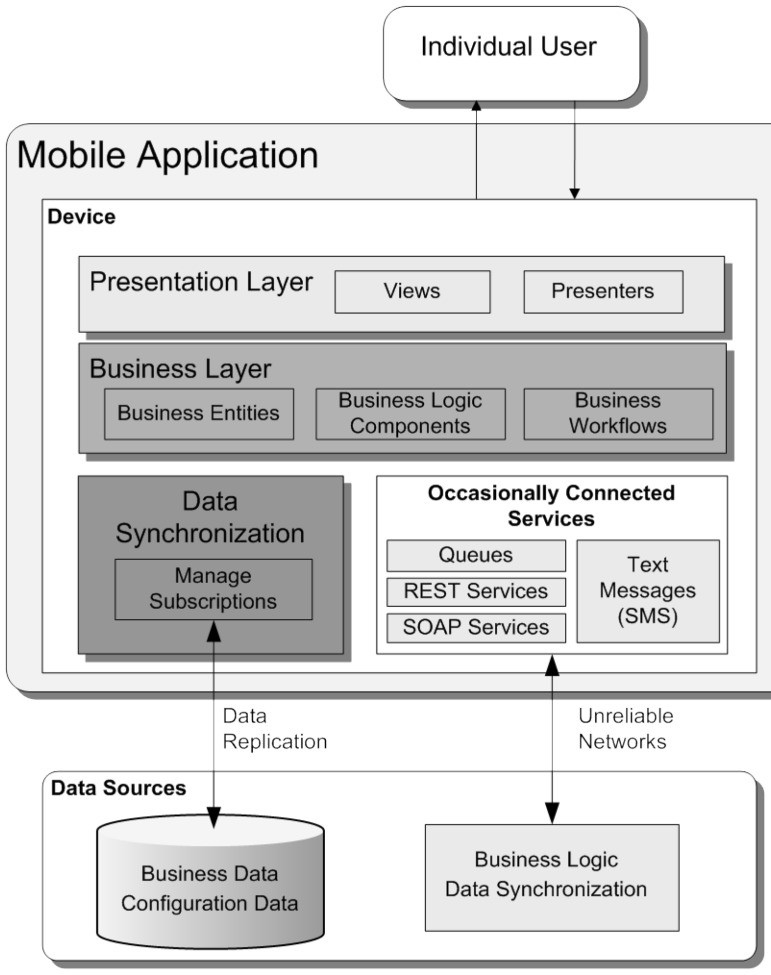

Figure 1 Common rich client mobile application architecture

Design Considerations

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

- **Decide if you will build a rich client, a thin Web client, or rich Internet application (RIA)**. If your application requires local processing and must work in an occasionally connected scenario, consider designing a rich client. Keep in mind that a rich client application will consume more device resources and will be more complex to install and maintain. If your application can depend on server processing and will always be fully connected, consider designing a thin client. If your application requires a rich user interface (UI), only limited access to local resources, and must be portable to other platforms, design an RIA client.
- **Determine the device types you will support**. When choosing which device types to support, consider screen size, resolution (DPI), CPU performance characteristics, memory and storage space, and development tool environment availability. In addition, factor in user requirements and organizational constraints.

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

Mobile Client Frame

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

Table 1 Mobile Client Frame

Category	Key issues
Authentication and Authorization	• Failing to authenticate in occasionally connected scenarios • Failing to authorize in occasionally-connected scenarios • Failing to use authentication and authorization over a virtual private network (VPN) • Failing to authenticate during synchronization over the air • Failing to authenticate during synchronization with the host PC

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

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

Authentication and Authorization

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

Consider the following guidelines when designing authentication and authorization:

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

Caching

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

Consider the following guidelines when designing caching:

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

Communication

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

Consider the following guidelines when designing your communication strategy:

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

Configuration Management

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

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

- Design for the restoration of configuration after a device reset.
- If you have you enterprise data in Microsoft SQL Server 2005 or 2008 and desire an accelerated time to market, consider using merge replication with a “buy and configure” application from a third party. Merge replication can synchronize data in a single operation regardless of network bandwidth or data size.
- Due to memory limitations, choose binary format over Extensible Markup Language (XML) for configuration files
- Protect sensitive data in device configuration files.
- Consider using compression library routines to reduce the memory requirements for configuration and state information.
- If you have a Microsoft Active Directory® directory service infrastructure, consider using the System Center Mobile Device Manager interface to manage group configuration, authentication, and authorization of devices. See the Technology Considerations section for requirements for the Mobile Device Manager.

Data Access

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

Consider the following guidelines when designing data access:

- Program for data integrity. Files left open during device suspend and power failures may cause data-integrity issues, especially when data is stored on a removable storage device.
- Include exception handling and retry logic to ensure that file operations succeed.
- Do not assume that removable storage will always be available, as a user can remove it at any time. Check for the existence of a removable storage device before writing or using FlushFileBuffers.
- If you need to ensure data integrity in case the device loses power or has connectivity disruptions, considering using transactions with SQL Server Mobile.
- If your application must access a disconnected database, consider using the device database services (such as SQL Server Compact Edition).
- If you use XML to store or transfer data, consider its overall size and impact on performance. XML increases both bandwidth and local storage requirements. Use compression algorithms or a non-XML transfer method.
- Consider the use of custom objects instead of **DataSets** to reduce memory overhead and improve performance.
- If your application needs to sync with multiple database types, use Sync Services for ADO.NET. It allows you to store data in Microsoft SQL Server, Oracle, or DB2.

Debugging

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

Consider the following guidelines when designing your debugging strategy:

- Understand your debugging costs when choosing which devices to support. Factor in tools support, the cost of initial (and perhaps replacement) test devices, and the cost of software-based device emulators.
- If you have the device you are targeting, debug your code on the actual device rather than on an emulator.
- If the device is not available, use an emulator for initial testing and debugging. Consider that an emulator might run code slower than the actual device. As soon as you obtain the device, switch to running code on the device connected to a normal PC. Perform final testing on your device when not connected to a PC. Add in temporary or permanent mechanisms to debug problems in this scenario. Consider the needs of people who will support the device.

- Test scenarios where your device is fully disconnected from any network or connection, including a PC debugging session.
- If you are an OEM or ODM and your device has not yet been created, note that it is possible to debug a mobile program on a dedicated x86-based Microsoft Windows® CE PC. Consider this option until your device is available.

Device

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

Consider the following guidelines when determining your device strategy:

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

Exception Management

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

Consider the following guidelines when designing for exception management:

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

Logging

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

Consider the following guidelines when designing logging:

- If you carry out extensive logging on the device, consider logging in an abbreviated or compressed format to minimize memory and storage impact. There is no system Event Log in Windows Mobile.
- If you do not require extensive logging, consider using OpenNetCF on the device. Be aware that OpenNetCF relies upon XML and will take up more space than a simple binary logging model.
- Consider using platform features such as health monitoring on the server, and mobile device services on the device, to log and audit events.
- Synchronize between the mobile database logs and the server database logs to maintain audit capabilities on the server. If you have an Active Directory infrastructure, consider using the System Center Mobile Device Manager to extract logs from mobile devices. See the Technology Considerations section for requirements for the Mobile Device Manager.
- Do not store sensitive information in log files.
- Decide what constitutes unusual or suspicious activity on a device, and log information based on these scenarios.

Porting

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

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

- If you are porting a rich client application from the desktop, rewrite the application in its entirety. Rich clients are rarely designed to suit a small screen size and limited memory and disk resources.
- If you are porting a Web application to a mobile device, consider rewriting the UI for the smaller screen size. Also, consider communication limitations and interface chattiness as these can translate into increased power usage and connection costs for the user.
- If you are porting an RIA client, research details to discover which code will port without modification. Consult the technology considerations section of this chapter for specific advice.

- Research and utilize tools to assist in porting. For example, Java-to-C++ convertors are available. When converting from Smartphone to Pocket PC code, Microsoft Visual Studio® will allow you to change the target platform and will provide warnings when you are using Smartphone-specific functionality. You can also link Visual Studio Desktop and Mobile projects to assist in knowing what is portable between the two projects.
- Do not assume that you can port custom controls as-is to a mobile application. Supported APIs, memory footprint, and UI behavior are different on a mobile device. Test the controls as early as possible so that you can plan to rewrite them or find an alternative if required.

Power

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

Consider the following guidelines when designing for power consumption:

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

Synchronization

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

Consider the following guidelines when designing synchronization:

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

User Interface

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

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

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

Performance Considerations

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

Consider the following guidelines when designing your performance strategy:

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

Deployment

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

Deployment scenarios listed for Windows Mobile device applications, with the more common ones listed first, are:

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

Consider the following guidelines when designing your deployment strategy:

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

Pattern Map

Table 2 Pattern Map

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

Pattern Descriptions

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

- **Reliable Sessions**. End-to-end reliable transfer of messages between a source and a destination, regardless of the number or type of intermediaries that separate the endpoints
- **Synchronization**. A component installed on a device tracks changes to data and exchanges information with a component on the server when a connection is available.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

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

Microsoft Silverlight for Mobile

If you are using Microsoft Silverlight® for Mobile, consider the following guidelines:

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

.NET Compact Framework

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

- If you are familiar with the Microsoft .NET Framework and are developing for both the desktop and mobile platforms concurrently, consider that the .NET Compact Framework is a subset of the .NET Framework class library. It also contains some classes exclusively

designed for Windows Mobile. The .NET Compact Framework supports only Microsoft Visual Basic® and Microsoft Visual C#® development.
- If you are attempting to port code that uses Microsoft Foundation Classes (MFC), consider that it is not trivial due to MFC's dependency on Object Linking and Embedding (OLE). The Windows Compact Edition supports COM, but not OLE. Check to see if the OLE libraries are available for separate download to your device before trying to use MFC on a mobile device.
- If you have issues tracing into a subset of Windows Mobile code with the Visual Studio debugger, consider that you might require multiple debug sessions. For example, if you have both native and managed code in the same debug session, Visual Studio might not follow the session across the boundary. In this case, you will require two instances of Visual Studio running and will have to track the context between them manually.

Windows Mobile

Consider the following general guidelines for Windows Mobile applications:
- If you are targeting an application for both Windows Mobile Professional and Windows Mobile Standard editions, consider that the Windows Mobile security model varies on the different versions of Windows Mobile. Windows Mobile Standard (Smartphone) supports three levels: blocked, normal, and privileged mode. Windows Mobile Professional (Pocket PC) supports only blocked and privileged mode. Code that works on one platform might not work on the other because of the differing security models for APIs. Check the Windows Mobile documentation for your device and version.
- If you will have to manage your application in the future or are upgrading an existing application, be sure that you understand the Windows Mobile operating system derivation, product naming, and versioning tree. There are slight differences between each version that could potentially impact your application.
    - Windows Mobile is derived from releases of the Windows CE operating system.
    - Both Windows Mobile version 5.x and 6.x are based on Windows CE version 5.x.
    - Windows Mobile Pocket PC was renamed Windows Mobile Professional starting with Windows Mobile 6.0
    - Windows Mobile Smartphone was renamed Windows Mobile Standard starting with Windows Mobile 6.0.
    - Windows Mobile Professional and Windows Mobile Standard have slight differences in their APIs. For example, the Windows Mobile Standard (Smartphone) lacks a Button class in its Compact Framework implementation because softkeys are used for data entry instead.
- Always use the Windows Mobile APIs to access memory and file structures. Do not access them directly after you have obtained a handle to either structure. Windows CE version 6.x (and thus the next release of Windows Mobile) uses a virtualized memory model and a different process execution model than previous versions. This means that structures such as file handles and pointers may no longer be actual physical pointers to memory. Windows

Mobile programs that relied on this implementation detail in versions 6.x and before will fail when moved to the next version of Windows Mobile.

- The Mobile Device Manager is mentioned in this article as a possible solution for authorizing, tracking, and collecting logs from mobile devices, assuming that you have an Active Directory infrastructure. MDM also requires a number of other products to fully function, including:
    - Windows Mobile 6.1 on devices
    - Windows Software Update Service (WSUS) 3.0
    - Windows Mobile Device Management Server
    - Enrollment Server
    - Gateway Server
    - Active Directory as part of Windows Server
    - SQL Server 2005 or above
    - Microsoft Certificate Authority
    - Internet Information Server (IIS) 6.0
    - .NET Framework 2.0 or above

Windows Embedded

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

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

Additional Resources

- For more information on the Windows Embedded technology options, see the Windows Embedded Developer Center at http://msdn.microsoft.com/en-us/embedded/default.aspx.
- For the patterns & practices Mobile Client Software Factory, see http://msdn.microsoft.com/en-us/library/aa480471.aspx
- For information on the Microsoft Sync Framework, see http://msdn.microsoft.com/en-us/sync/default.aspx
- For more information on the OpenNETCF.Diagnostics.EventLog in the Smart Device Framework see http://msdn.microsoft.com/en-us/library/aa446519.aspx

- For more information on ASP.NET Mobile, see http://www.asp.net/mobile/road-map/
- For more information on adding ASP.NET Mobile source code support into Visual Studio 2008, see http://blogs.msdn.com/webdevtools/archive/2007/09/17/tip-trick-asp-net-mobile-development-with-visual-studio-2008.aspx

Chapter 20: Office Business Applications (OBA)

Objectives

- Define an Office Business Application (OBA).
- Understand the common application types where OBAs are suitable.
- Understand the architecture of OBAs and their integration with Microsoft® Office SharePoint® and line-of-business (LOB) applications.
- Understand the components found in an OBA.
- Learn the key scenarios for OBAs.
- Learn the design considerations for OBAs.
- Learn the key patterns associated with OBAs.

Overview

Office Business Applications (OBAs) are a class of enterprise composite applications. They provide solutions that integrate the core capabilities of back-end business systems with the widely deployed and widely used business productivity services and applications that constitute the Microsoft Office System. OBAs implement business logic that is maintained through end-user forms, providing a rich user experience that can help to improve business insight and assist in integrating existing internal or external systems.

OBAs usually integrate with new and existing line-of-business (LOB) applications. They leverage the rich user interface (UI) and automation capabilities of the Office clients to simplify complex processes that require user interaction, and help to minimize errors and improve processes. Effectively, OBAs use the Office client applications to fill the gaps between existing LOB systems and users.

Architecture

Figure 1 illustrates the key components and layers of an OBA. One thing to note is that this diagram adds a layer named Productivity between the Presentation and Application Services Layers. The Productivity layer contains components used to store and manage collaborative work streams in a document-centric manner.

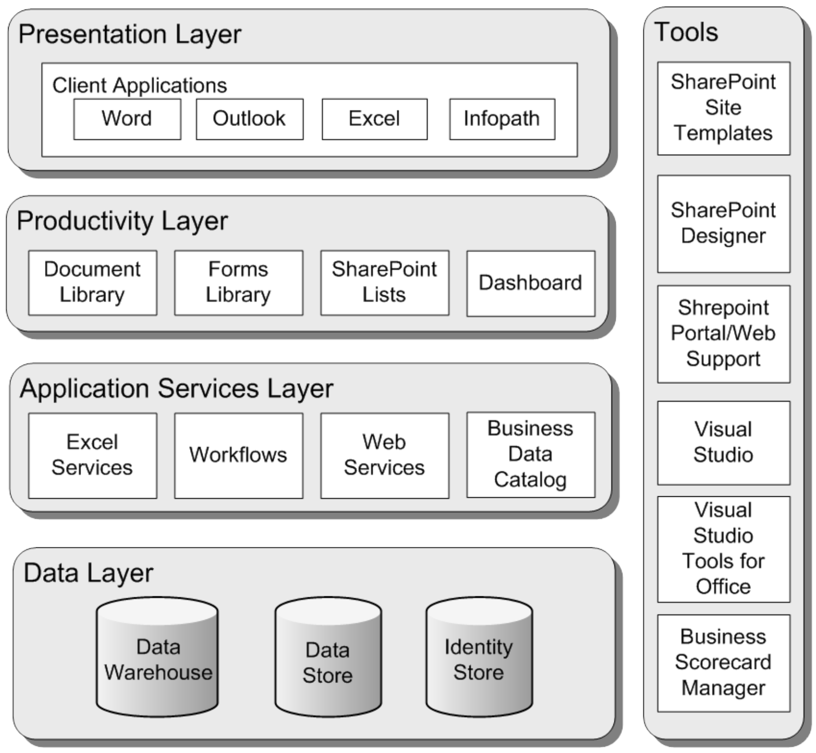

Figure 1 Key components of an OBA

OBAs are designed to interoperate using standard file formats and Web services; for example:
- Open standards and interoperability are the core tenets of the Microsoft Office System.
- The metadata definitions of OBA solution objects are based on Extensible Markup Language (XML) schemas.
- All Office products are service-enabled at all levels.
- Interoperable OpenXML file formats are the default schemas for business documents created by or generated for Microsoft Office business productivity client applications.

Key Components

An OBA is made up of a variety of applications and services that interact to provide an end-to-end solution to a business problem. It may contain or be created using any or all of the following items:
- **Microsoft Office clients**. The client applications include Microsoft Office Outlook®, Microsoft Office Word, Microsoft Office Excel®, Microsoft Office InfoPath®, and Microsoft Office PowerPoint®. Custom forms in Outlook can be used to host UI controls with the ability to integrate business logic and data from various sources. Word and Excel offer programmability in the form of the Task Pane, Smart Tags, and the new Ribbon. This makes it possible to combine natural document interactions with structured business data and processes. Smart Tags use regular expression pattern-matching to recognize identifiers such as telephone numbers, government identification numbers, or custom account numbers within the text of a document. Relevant actions can be presented in the document alongside the data.

- **Microsoft Windows® SharePoint Services (WSS)**. Built on Microsoft Windows Server® 2003, WSS provides content-management and collaboration features that can help improve business processes and team productivity. WSS also provides a platform for building Web-based business applications that allow users to share documents, information, and ideas with support for offline synchronization and task management.
- **Microsoft Office SharePoint Server (MOSS)**. MOSS extends the capabilities provided by WSS to offer enterprise-wide functionality for content management, workflow, search, portals, and personalized sites. In addition, MOSS provides Excel Services for reporting, the Business Data Catalog (BDC) for LOB access, and a security framework for single-sign-on (SSO) capabilities.
- **Technologies and services**. Excel Services allow documents to be authored by clients using Excel in the usual way, and then saved to SharePoint Server. End users can view and interact with the documents in a Web browser, and software developers can programmatically invoke business logic stored within the documents. Windows Workflow Foundation (WF) functionality is built into MOSS. This makes it easy to capture a process, such as a purchase order approval, and reduce user errors and associated delays. The ASP.NET run time supports Web Page and Web Part rendering to create customized Web sites that reflect the company's requirements.
- **Collaboration features.** Collaboration can be managed by Microsoft Office Communications Server (OCS), Microsoft Office Groove® Server, and Microsoft Exchange Server.
- **Development tools**. These include SharePoint Central Administration, SharePoint Designer, Microsoft Visual Studio®, and Visual Studio Tools for Office (VSTO).

Key Scenarios

OBAs generally fall into one of three categories that implement key scenarios. These categories are:

- **Enterprise content management** allows people to find and use role-based information.
- **Business intelligence** enables business insight through capabilities such as server-based Excel solutions.
- **Unified messaging** enables communication, and collaboration, which simplifies team management.

Enterprise Content Management

Enterprise content management scenarios allow people to find and use role-based information. One of the more common scenarios in business environments is the use of MOSS or WSS as a content-management tool for Office client documents. With either of these SharePoint solutions, you can implement versioning and workflow on the files associated with Office client applications. In addition, many of the files can be modified within the SharePoint environment, and features included with MOSS use Microsoft Office Excel to create and display reports. As a result, many of the key scenarios are based on using SharePoint with Microsoft Office client applications.

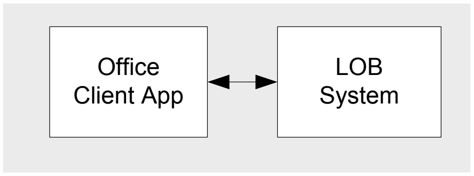

Figure 2a Office client interacting directly with an LOB system

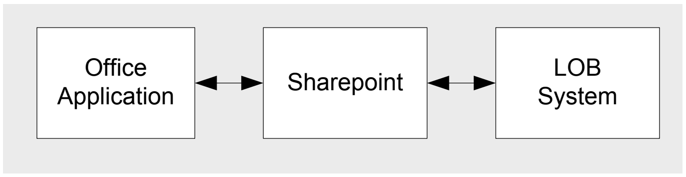

Figure 2b Office client interacting with LOB system through a SharePoint intermediary

The following OBA patterns, described in detail later in this chapter, are also useful for implementing enterprise content management scenarios:

- The Extended Reach Channel pattern extends LOB application functionality to a broader user base using Office applications as the channel.
- The Document Workflow pattern enables control and monitoring of document-centric processes, and can infuse best practices and enhance underlying business processes.
- The Collaboration pattern augments structured business processes with unstructured human collaboration.

Business Intelligence

Business intelligence scenarios enable business insight through capabilities such as server-based Excel solutions. The following OBA patterns, described in detail later in this chapter, are useful for implementing business intelligence scenarios:

- The Document Integration pattern enables the generation of Office documents from LOB applications; enables information workers to embed LOB data in Office documents by interacting with LOB data while authoring the document; and enables server-side processing of documents containing LOB data.
- The Composite UI pattern supports composition of multiple application UIs in an Office document or a SharePoint Web page.
- The Data Consolidation pattern enables a more natural way of interacting with LOB data by allowing users to discover data using searches across multiple LOB applications, and then act on the results. Data Consolidation uses the Discovery Navigation pattern.

Unified Messaging

Unified messaging scenarios support communication and collaboration, which simplifies team management. The Notification and Tasks pattern, described in detail later in this chapter, is useful for implementing unified messaging scenarios. The Notification and Tasks pattern uses Outlook as a primary UI to receive and act on LOB application–generated tasks and alerts.

Summary of Common OBA Patterns

OBAs can vary from the very simple to extremely complex custom solutions. OBAs generally incorporate one or more of the following common patterns:

- **Extended Reach Channel**. These applications extend LOB application functionality to a broader user base using Office applications as the channel.
- **Document Integration**. These applications enable the generation of Office documents from LOB applications; enable information workers to embed LOB data in Office documents by interacting with LOB data while authoring the document; and enable server-side processing of documents containing LOB data.
- **Document Workflow**. These applications enable control and monitoring of document-centric processes, and can infuse best practices and enhance underlying business processes.
- **Composite UI**. These applications support composition of multiple application UIs in an Office document or a SharePoint Web page.
- **Data Consolidation**. These applications enable a more natural way of interacting with LOB data by allowing users to discover data using searches across multiple LOB applications, and then act on the results.
- **Collaboration**. These applications augment structured business processes with unstructured human collaboration.
- **Notifications and Tasks**. These applications use Outlook as a primary UI to receive and act on LOB application–generated tasks and alerts.

Large and complex solutions may incorporate more than one of these patterns, or may use patterns different from those shown. However, these patterns will help you to think about how you design an OBA. Later sections of this chapter explore each of these patterns in more depth.

Design Considerations

The design of a suitable OBA is based on the scenarios you must support, and the types of Office client applications suitable for those scenarios. In addition to considering the base patterns shown in the previous section, consider the following guidelines when designing your OBA:

- **Consider using a mediated integration pattern over direct integration**. When designing an OBA as an extended-reach channel, you can implement interfaces directly within documents. For example, an Excel spreadsheet can contain custom input forms. However, this approach requires custom code and limits your ability to reuse functionality. With a mediated integration pattern, you can take advantage of applications such as SharePoint and the Business Data Catalog to decouple the interfaces from the physical documents.
- **Use OpenXML-based schemas for embedding LOB data in documents**. OpenXML is a European Computer Manufacturers Association (ECMA) international standard that is supported by Office 2007 applications, as well as by many independent vendors and platforms. By using OpenXML, you can share data between Office applications and applications developed for other platforms.
- **Create LOB document templates for common layouts that will be reused**. An LOB template contains markup and metadata associated with the LOB that can be bound to specific LOB

data instances at a later time. In other words, new documents can be generated by merging LOB data with document templates. End users can create custom documents without developer involvement, and complex documents can be generated using server-side batch processing.

- **Use MOSS to control the review and approval process for documents**. Microsoft Office SharePoint Server (MOSS) provides out-of-the box features that support a basic workflow process for the review and approval of documents. For more complex processing requirements, Windows Workflow Foundation (WF) can be used to extend the workflow capabilities found in SharePoint.
- **Use the Collaboration pattern for human collaboration**. Most LOB applications are good at handling structured business processes. However, they are not good at handling the unstructured nature of human interaction with business processes. A site implementing the collaboration pattern addresses this issue by providing an interface geared toward collaboration with other users. The SharePoint Team Site template implements this pattern.
- **Consider remote data-synchronization requirements**. Documents that are created, updated, or distributed should be synchronized with the LOB system and then stored for future use. Even though LOB systems are quite useful for handling transaction-oriented activities, they are not suited to capturing the significant work that occurs between activities.

OBA Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into the base OBA patterns listed and described in this chapter. The following table lists the common issues for each pattern where mistakes are most often made.

Table 1 OBA frame

Category	Key issues
Extended Reach Channel	• Duplication of functionality across an enterprise • Stand-alone applications with limited reuse
Document Integration	• Not using open standards for embedding LOB data • Creating common layouts by hand for each new document
Document Workflow	• Not considering workflow requirements • Building custom workflow components instead of using workflow capabilities in SharePoint
Composite UI	• Not following Office standards • Creating custom components when Web Parts that provide the required functionality are available
Data Consolidation	• Not providing sufficient LOB entity data for Office applications to act on
Collaboration	• Not considering the unstructured nature of human collaboration
Notifications & Tasks	• Using multiple LOB applications to provide task and notification support

Extended Reach Channel

Extended Reach Channel applications extend LOB application functionality to a broader user base using Office applications as the channel. The Extended Reach Channel pattern is useful for implementing the following scenarios:

- Eliminating duplication of effort that currently exists in your enterprise, such as an Outlook feature for consultants to assign time for meetings to billable projects.
- Extending LOB functionality to a broader set of end users, such as a self-service application that allows employees to update their personal information.
- Improving the use of an existing system that users currently avoid because of duplication of effort, or lack of training.
- Collecting information from users through e-mail and automatically updating the system.

The Extended Reach Channel approach supports two different integration patterns: the Direct Integration pattern and the Mediated Integration pattern. The following sections describe these patterns.

Direct Integration Pattern

The Direct Integration pattern is where Office client applications expose LOB functionality directly to a broader set of users. In this pattern, access to LOB interfaces is projected directly into an Office client or is extended to an existing behavior such as calendaring. The client application may access the LOB data through a Web service.

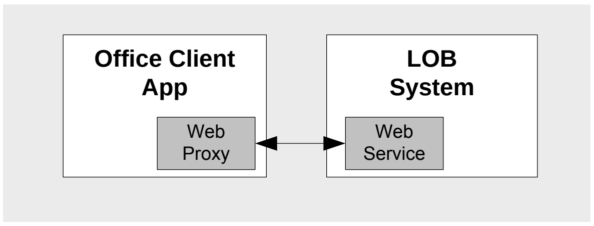

Figure 3a The Direct Integration pattern using Web services

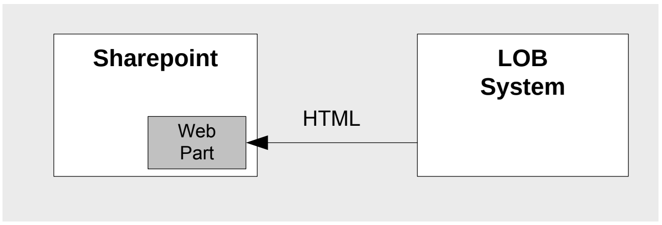

Figure 3b The Direct Integration pattern using HTML

Mediated Integration Pattern

The Mediated Integration pattern is where metadata stores such as the Business Data Catalog (BDC) are used to provide an additional level of abstraction that provides common approaches to managing LOB documents, including security with a single sign-on mechanism based on a credentials mapping.

This pattern provides more opportunities for composing services and data into a composite UI. A mediator, which could be the BDC, collects data from disparate sources and exposes it in Office-compatible formats and services that client applications can consume. Figure 5 illustrates the Mediated Integration pattern.

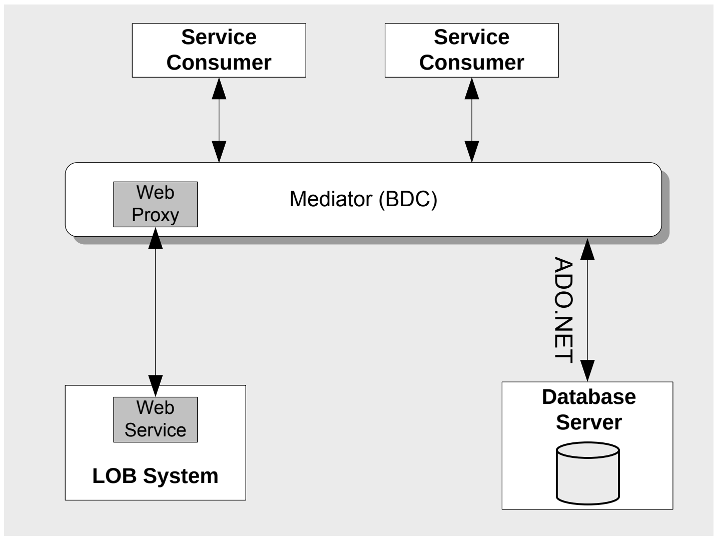

Figure 4 The Mediated Integration pattern

Document Integration

Document Integration applications enable the generation of Office documents from LOB applications; enable information workers to embed LOB data in Office documents by interacting with LOB data while authoring the document; and enable server-side processing of documents containing LOB data. The Document Integration pattern is useful for implementing the following scenarios:

- Reducing duplication of LOB data that is stored in individual Office documents located on user desktop systems.
- Exposing specific subsets of LOB data to Office applications for tasks such as mail merge or reporting.
- Generating Office documents that include items of LOB data in the appropriate format, automatically refreshed as the data changes.
- Generating documents that require custom server-side processing of LOB data.
- Accepting inbound documents, processing the embedded data, and applying it to the LOB system.

The Document Integration approach supports four different integration patterns that use XML to pass information to and from LOB systems. The simplest is the Application Generated Documents pattern. In addition, there are three Intelligent Document integration patterns: the Embedded LOB Information pattern, the Intelligent Documents: Embedded LOB Template pattern, and the Intelligent Documents: LOB Information Recognizer pattern. The following sections describe these patterns.

Application Generated Documents Pattern

The Application Generated Documents pattern is where the LOB system merges business data with the Office document using batch-oriented server-side processing, although client-side generation is also feasible. Common examples include exporting data to Excel spreadsheets, or generating reports and letters in Word. This is the most commonly used pattern for document data integration.

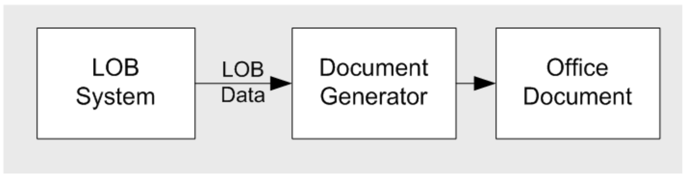

Figure 5 The Application Generated Documents pattern

Intelligent Documents: Embedded LOB Information Pattern

The Intelligent Documents: Embedded LOB Information pattern is where LOB data is embedded directly in the body of the Office document, or embedded as an XML document part and exposed through a content control. Alternatively, the Office application can use the Office Custom Task Pane (CTP) to display LOB data that an information worker can browse or search, and embed into a document. Figure 6 illustrates the Embedded LOB Information pattern.

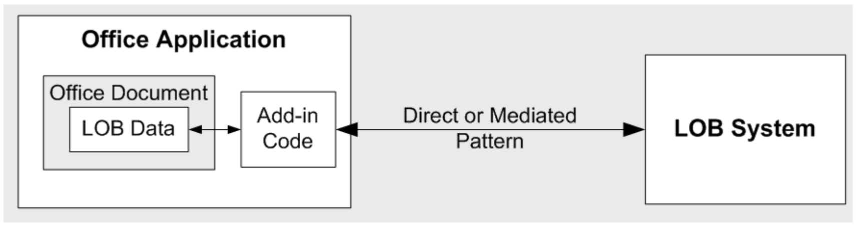

Figure 6 The Embedded LOB Information pattern

Intelligent Documents: Embedded LOB Template Pattern

The Intelligent Documents: Embedded LOB Template pattern is where a template is used to combine metadata from an LOB system with document markup, such as content controls, XML schemas, bookmarks, named ranges, and smart tags. At run time, the template is merged with appropriate instances of the LOB data to create a document. The merging can take place through an add-in within the Office client application, or on the server.

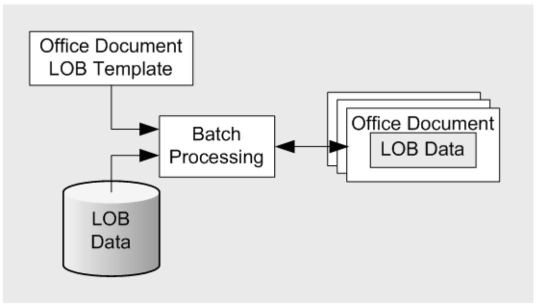

Figure 7 The Embedded LOB Template pattern

Intelligent Documents: LOB Information Recognizer Pattern
The Intelligent Documents: LOB Information Recognizer pattern is where metadata and document markup, such as content controls, XML schemas, bookmarks, named ranges, or smart tags contain data recognized by the LOB system. The application can use this data to update the LOB system, or to provide extra functionality for users. On the server side, the application may start a workflow using the information. On the client, the application might present context-sensitive information, such as the details of a customer whose name is recognized in a Word document.

Document Workflow
Document Workflow applications enable control and monitoring of document-centric processes, and can infuse best practices and enhance underlying business processes. The Document Workflow pattern is useful for implementing the following scenarios:
- Applications that exchange information, often via e-mail, to perform multistep tasks such as forecasting, budgeting, and incident management
- Applications where specific legal or corporate compliance procedures must be followed, and audit information maintained
- Applications that carry out complex document-handling and conditional-routing tasks, or that must implement best practice–based on rules

The Document Workflow approach supports two different integration patterns that initiate workflows:
- **LOB Initiated Document Workflow pattern**, where documents are passed to a SharePoint document workflow automatically by an action such as saving them to a SharePoint document library, or submitting an InfoPath form. The workflow might send the document to the next recipient in a list, store copies, or perform processes on the document depending on the requirements of the application.
- **Cooperating Document Workflow pattern**, where there may be a series of interactions between documents and LOB systems that must follow certain rules or prevent certain actions; for example, preventing edits to a submitted document at a specific stage of the process, extracting specific information, and publishing this information back to the LOB system. This pattern will usually use a SharePoint cooperating workflow that provides the flow logic, while the intelligent document provides the LOB interaction mechanisms. In complex scenarios, the LOB system may also update the document as it passes through the workflow.

Composite UI
Composite UI applications support composition of multiple application UIs in an Office document or a SharePoint Web page. The Composite UI pattern is useful for implementing the following scenarios:
- Applications that collect and display several different types of information in a single UI page or screen

- Applications that use data exposed by multiple back-end systems, and display it in a single UI page or screen
- Applications that must provide a customizable composite interface that users modify to best suit their requirements

The Composite UI approach supports several different integration patterns that combine information into a composite UI:

- **Context Driven Composite User Interface pattern**, where contextual information determines the UI composition. The contextual information can be static (such as the application configuration, or adding a tab to the Outlook view) or dynamic (such as hiding or showing tab-based data in the source document). Each region of the composite UI presents information through an Office client component. However, users cannot dynamically change the linking at run time between the document components and the source data located in the LOB system.
- **Mesh Composite View pattern**, where the UI contains components such as ASP.NET Web Parts or MOSS components that cooperatively interact to expose data from the same or different LOB systems. For example, a part that represents a view of a customer from a customer relationship management (CRM) system might be connected at the time the view is constructed to a part that represents a list of open order status in an enterprise resource planning (ERP) system. When a customer is selected in the CRM part, the CRM part raises an event and provides the information on the selected customer identity to the open order status part, which in turn displays the list of order status for the selected customer.
- **RSS and Web Services Composition pattern**, which is a specialized version of the Mesh Composite View pattern that combines data published as RSS feeds or through Web services. Multiple SharePoint Data View Web Parts (or custom parts) format and present the published data within the UI. An example is a composite view of the catalogs of several suppliers, where each published item provides a link to a page on the supplier’s Web site that contains extra information.
- **Analytics pattern**, which is a specialized version of the Mesh Composite View pattern that presents a data analysis dashboard to the end user. It can use Excel Services and the Excel Services Web Part provided by MOSS 2007 to display data and charts, or other parts to display custom data and information from the LOB system, and from other sources, within the composite UI. A useful part provided by MOSS for dashboards is the Key Performance Indicator (KPI) Web Part that allows users to define KPIs based on data in any SharePoint list, including a BDC list.

Data Consolidation (Discovery Navigation)

Data Consolidation applications enable a more natural way of interacting with LOB data by allowing users to discover data using searches across multiple LOB applications, and then act on the results. Data Consolidation uses the Discovery Navigation pattern. The Discovery Navigation pattern is useful for implementing the following scenarios:

- Applications that provide search capabilities for a single LOB system
- Applications that provide search capabilities across multiple LOB systems

- Applications that provide search capabilities across a diverse range of LOB systems and other data sources

Data Consolidation Pattern

The Data Consolidation pattern provides a consistent search experience for information workers by combining the results of searches over one or more sources into a single result set, and presenting not only Uniform Resource Identifiers (URIs) that link to the results, but also actions associated with the found items. Figure 8 illustrates the Data Consolidation pattern creating a content index.

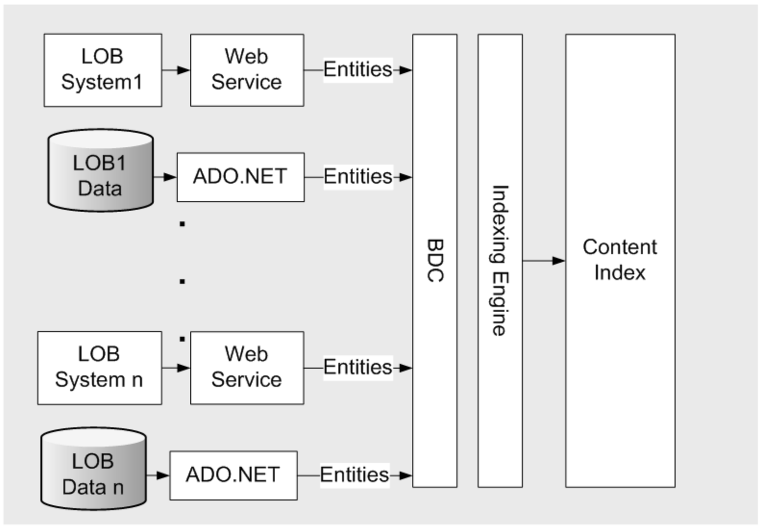

Figure 8 The content index contains information collated from a range of sources

Launching an LOB Process

The action links can initiate an LOB operation, such as starting a workflow or performing a process on a document, as illustrated in Figure 9.

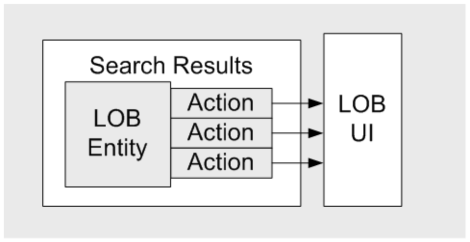

Figure 9 Launching an LOB process based on an action for an item in the search results

Collaboration

Collaboration applications augment structured business processes with unstructured human collaboration. The Collaboration pattern is useful for implementing the following scenarios:

- Applications that involve human interaction that leads to interaction with a LOB system, such as discussion of a sales opportunity before committing an order

- LOB applications that collate content and user contributions in an unstructured form and later need to use it in a structured format
- Applications that provide information in an unstructured form that users may be able to edit, such as a wiki or discussion site

The Collaboration pattern uses MOSS Team Site templates that allow users to collaborate around a specific business problem, using document libraries, discussion and task lists, team calendars, and simple project-management features. The site can be provisioned and populated using LOB data, and exposes links to LOB processes within the appropriate libraries and lists. Access can be through Office documents, or a Web browser.

Notifications and Tasks

Applications that need to support notifications and tasks use Outlook as a primary UI to receive and act on LOB application–generated tasks and alerts. In addition to Outlook, SharePoint provides notification and task services that can interact with most e-mail systems using the Simple Mail Transfer Protocol (SMTP). The Notifications and Tasks pattern is useful for implementing the following scenarios:

- Applications that assign tasks and generate notifications for end-users
- Applications that integrate multiple LOB operations and must notify users of status or process requirements

The e-mail–based Notifications and Tasks approach supports several different integration patterns that can notify users of tasks and status:

- **Simple Task & Notification Delivery pattern**, where the LOB system delivers tasks and notifications to users as Outlook tasks and e-mail messages in a one-way flow of information. Details of the task or the notification are embedded in the body of the task and e-mail message, but changes are not reflected back in the LOB system. Options for delivering tasks and notifications include delivering them to Microsoft Exchange Server (the push model), using an add-in on Outlook that fetches them (the pull model), or publishing an RSS feed to which users can subscribe.
- **Direct Task Synchronization pattern**, where the LOB system sends tasks to users via Exchange or Outlook in a synchronized bidirectional flow of information. Users and the LOB can update tasks at any time, and the changes are propagated to the LOB system. The task may be part of an LOB workflow.
- **Mediated Task Synchronization pattern** is a variant of the Direct Task Synchronization pattern, where MOSS acts as a mediator between the LOB system and Outlook in order to synchronize tasks. The LOB system publishes tasks to a SharePoint Task List, which is synchronized with Outlook Tasks by using Outlook's native synchronization mechanism. Updates to the task in Outlook are automatically pushed back to SharePoint, which raises an event indicating that the change has occurred and allows custom code to update the LOB system.
- **Intelligent Tasks & Notifications pattern**, where action links located in the Outlook Custom Task Pane (CTP) allow users to initiate specific actions based on the tasks or notifications

sent by the LOB system. Common tasks involve automatically logging on to the LOB system, finding the right information, and updating it. An example is a manager viewing an e-mail message sent by Human Resources to approve a vacation request for an employee, where the CTP contains action links that allow the manager to approve or reject the request by updating the LOB system.

- **Form-based Tasks & Notifications pattern** is a variant of the Intelligent Tasks & Notification pattern, where the e-mail message contains an attached InfoPath form pre-populated by the LOB system. The user can open the e-mail message, fill out the form, and submit it to the LOB system. InfoPath provides data validation, custom calculations, and logic to assist the user when filling out the form. The InfoPath CTP can provide additional information, extracted from the LOB system, to assist the user. A variant of this pattern uses MOSS InfoPath Forms Services to allow users to fill out forms in a Web browser without requiring InfoPath to be installed.

Security Considerations

Security is important in Office Business Applications that expose data and functionality through several types of client applications, and have access to corporate LOB data. It is important to secure all access to resources, and to protect data passing over the network. Consider the following guidelines for security when creating OBAs:

- Consider implementing single- sign-on so that users access the client applications and the back-end functionality using their current logon credentials, or credentials validated through a federated service such as the Microsoft Active Directory® directory service or SharePoint.
- Consider encrypting messages that pass outside of your secure network where possible.
- Consider using channel encryption such as Internet Protocol Security (IPSec) to protect the network connection between servers and clients.
- Consider using the trusted subsystem model for data access using role credentials to minimize the number of connections required.
- Consider filtering data at the server to prevent exposure of sensitive data in client applications where this is not necessary.

Deployment Considerations

You can deploy OBA solutions using either a Windows Installer or the Click Once technology:

- **Click Once** installation requires little user interaction, provides automated updates, and requires little effort for the developer. However, it can only be used to deploy a single solution that is not part of a larger solution; it cannot deploy additional files or registry keys; it cannot interact with the user to configure the installation; and it cannot provide a branded installation.
- **Windows Installer** installation can deploy additional components and registry settings; can interact with the user to configure the installation; and supports custom branding of the installation. However, it requires advanced configuration, more developer effort, and cannot provide automated updates.

Pattern Map

Table 2 Pattern map

Category	Relevant patterns
Extended Reach Channel	• Direct Integration • Mediated Integration
Document Integration	• Application Generated Documents • Embedded LOB Information • Embedded LOB Template • LOB Information Recognizer
Document Workflow	• LOB Initiated Document Workflow • Cooperating Document Workflow
Composite UI	• Context Driven Composite User Interface • Mesh Composite View • RSS and Web Services Composition • Analytics
Data Consolidation	• Discovery Navigation
Collaboration	• Collaboration
Tasks & Notifications	• Simple Task & Notification Delivery • Direct Task Synchronization • Mediated Task Synchronization • Intelligent Tasks & Notifications • Form-based Tasks & Notifications

Pattern Descriptions

- **Analytics**. A specialized version of the Mesh Composite View pattern that presents a data analysis dashboard to the end user.
- **Application Generated Documents**. The LOB system merges business data with an Office document using batch-oriented server-side processing.
- **Collaboration**. Use unstructured human collaboration to augment structured business processes.
- **Context Driven Composite User Interface**. Use contextual information to determine the composition of the UI.
- **Cooperating Document Workflow**. A series of interactions between documents and LOB systems that must follow certain rules or prevent certain actions.
- **Direct Integration**. Access to LOB interfaces is projected directly into an Office client, or is extended to an existing behavior such as calendaring.
- **Direct Task Synchronization**. The LOB system sends tasks to users via Exchange or Outlook as a synchronized bidirectional flow of information.
- **Discovery Navigation**. Allow users to discover data by searching across multiple LOB applications, and then act on the results.

- **Embedded LOB Information**. LOB data is embedded directly in the body of the Office document, or embedded as an XML document part and exposed through a content control.
- **Embedded LOB Template**. A template combines metadata from an LOB system with document markup, such as content controls, XML schemas, bookmarks, named ranges, and smart tags.
- **Form-based Tasks & Notifications**. A variant of the Intelligent Tasks & Notification pattern, where the e-mail message contains an attached InfoPath Form pre-populated by the LOB system.
- **Intelligent Tasks & Notifications**. Action links located in the Outlook Custom Task Pane (CTP) allow users to initiate specific actions based on the tasks or notifications sent by the LOB system.
- **LOB Information Recognizer**. Metadata and document markup—such as content controls, XML schemas, bookmarks, named ranges, or smart tags—contain data recognized by the LOB system.
- **LOB Initiated Document Workflow**. Documents are passed to a SharePoint document workflow automatically by an action such as saving them to a SharePoint document library, or submitting an InfoPath form.
- **Mediated Integration**. A mediator, which could be the BDC, collects data from disparate sources and exposes it in Office-compatible formats and services that client applications can consume.
- **Mediated Task Synchronization**. A variant of the Direct Task Synchronization pattern, where MOSS acts as a mediator between the LOB system and Outlook in order to synchronize tasks.
- **Mesh Composite View**. Use components in the UI, such as ASP.NET Web Parts or MOSS components, which cooperatively interact to expose data from the same or different LOB systems.
- **RSS and Web Services Composition**. A specialized version of the Mesh Composite View pattern that combines data published as RSS feeds or through Web services.
- **Simple Task & Notification Delivery**. The LOB system delivers tasks and notifications to users as Outlook tasks and e-mail messages in a one-way flow of information.

Additional Resources

For more information, see the following resources:

- *Automating Public Sector Forms Processing and Workflow with Office Business Application* at http://blogs.msdn.com/singaporedpe/archive/tags/OBA/default.aspx.
- *Getting Started with Office Business Applications* at http://msdn.microsoft.com/en-us/library/bb614538.aspx.
- *OBA (Reference Application Pack) RAP for E-Forms processing* at http://msdn2.microsoft.com/en-us/architecture/bb643796.aspx.
- PowerPoint slides (accompanying source code to be available soon) from http://msdn2.microsoft.com/en-us/architecture/bb643796.aspx.
- *OBA Central* portal site at http://www.obacentral.com.

- *Integrating LOB Systems with the Microsoft Office System* at http://msdn.microsoft.com/en-us/architecture/bb896607.aspx.
- *Understanding Office Development* at http://msdn.microsoft.com/en-us/office/aa905371.aspx.

Chapter 21: SharePoint LOB Applications

Objectives

- Define a SharePoint line-of-business (LOB) application.
- Learn the key scenarios and design considerations for LOB applications.
- Learn the key components and services for SharePoint LOB applications.
- Learn deployment options for SharePoint LOB applications.

Overview

Microsoft® Office Business Applications (OBAs) can integrate LOB processes with rich user experiences for data access, data analysis, and data manipulation by using role-tailored business portals built on top of Microsoft Windows SharePoint® Services (WSS) and the Microsoft Office SharePoint Server (MOSS). WSS sites can be configured to publish Internet-facing content, and sites can scale out with Web farm deployment to service large numbers of users.

WSS integrates tightly with the broader Microsoft platform. Microsoft Windows Server® is the core operating system on which WSS runs. WSS uses Internet Information Services (IIS) as a front-end Web server to host and scale out Web sites. It uses Microsoft SQL Server® on the back end to store site definitions, content type definitions, published content, and configuration data.

WSS can also integrate with ASP.NET to provide LOB data presentation for sites. It can use ASP.NET Web Parts, styles, themes, templates, server controls, and user controls for the user interface UI).

Figure 1 shows the key features and layers of a SharePoint LOB application.

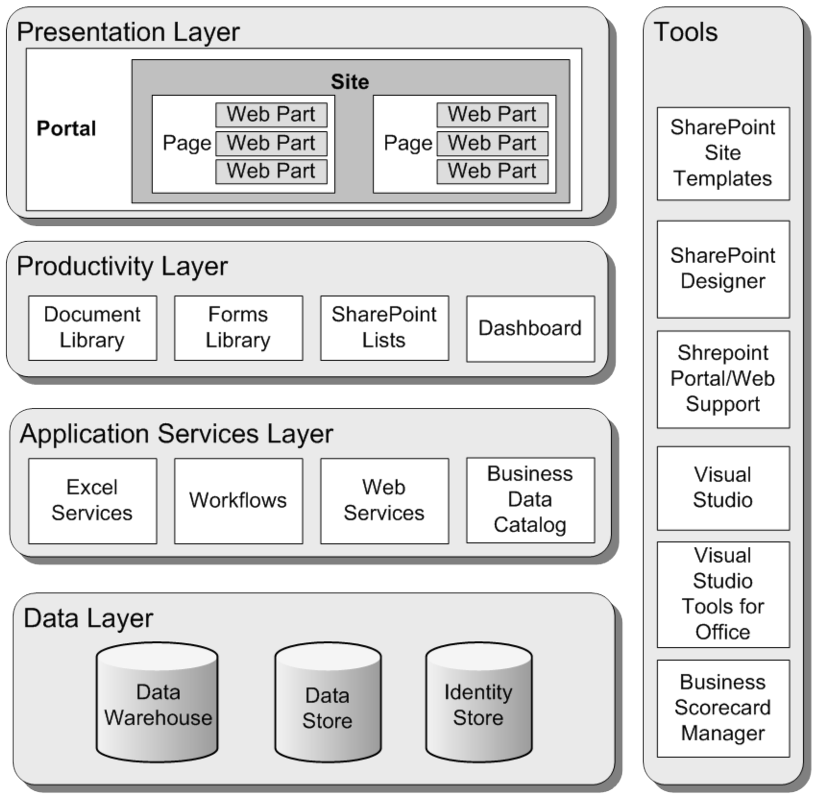

Figure 1 Key features of a SharePoint LOB application

Layers

The following list describes each of the layers of a SharePoint LOB application:

- **Presentation layer**. This is the UI of the application. Users connect through a browser to the SharePoint Server Portal, which is composed of Web pages. These Web pages can be assembled by using Web Parts, which provide rich composition at the presentation level. Web Parts for Office client applications are also available, and you can build custom Web Parts to implement application-specific functionality.
- **Productivity layer**. The Microsoft Office client applications are used for information processing and collaboration. Office documents, such as Microsoft Office Excel® spreadsheets, are stored in document libraries. Forms that automate tasks in the Office applications are stored in forms libraries. The productivity layer also implements features for creating and publishing reports, in the form of either SharePoint lists or Excel spreadsheets, by using Excel Services. It can also generate output in the form of a dashboard composed of information drawn from multiple services.
- **Application services layer**. This is a reusable layer within the application. Microsoft Office System applications can integrate with a service-oriented architecture (SOA). The Office system also supports workflows using Windows Workflow Foundation (WF), which can provide business process or document life-cycle management. The Office System clients can consume service interfaces by invoking the Web services directly, or by using the Business Data Catalog (BDC). Excel Services can also be used to build applications directly within the Office System.

- **Data layer**. This layer encapsulates the mechanisms for storing and accessing all the different types of data required by the application. This includes roles and identities, as well as the operations data and data warehouses that contain the LOB data.

Tiers

The previous section describes the logical grouping of components or functionality of a SharePoint LOB application into separate layers. You must also understand the physical distribution of components on separate servers of your infrastructure. The following list describes the common scenarios and guidelines:

- Deploy the databases for SharePoint on a separate database server or database cluster for maximum reliability and performance.
- In a non-distributed scenario, deploy the presentation, productivity, and application services layers on the same Web server or a Web farm.
- In a distributed scenario, you can deploy the components of the presentation layer (portals, sites, pages, and Web Parts) on a Web server or a Web farm and the remaining layers and components on a separate application server or application farm.
- For maximum performance under severe load, you might want to deploy the components for the application services layer on a separate application server or application farm.

Key Components

MOSS assists in providing content-management features and implementing business processes. SharePoint sites support specific content publishing, content-management, records-management, and business-intelligence needs. You can also conduct effective searches for people, documents, and data, participate in forms-driven business processes, and access and analyze large amounts of business data.

- **Workflow**. MOSS is integrated with WF, and allows developers to create simple workflows and attach them to the document libraries in SharePoint. Users can also create custom workflows using the SharePoint designer.
- **Business intelligence.** MOSS provides users with interactive Business Intelligence portals that support substantial data manipulation and analysis. Users can create dashboards from multiple data sources without writing code. Key Performance Indicators (KPIs) can be defined from Excel Services, SharePoint lists, SQL Server Analysis Services cubes, and a variety of other sources. Because this data is hosted with SharePoint, it can be an active participant in other SharePoint services such as search and workflow.
- **Content management**. Functionality from Microsoft Content Management Server (MCMS) has been rolled into MOSS, allowing it to take advantage of comprehensive Web content-management features available directly from the SharePoint platform.
- **Search**. Enterprise Search in MOSS is a shared service that provides extensive and extensible content gathering, indexing, and querying facilities, and supports full-text and keyword searches.
- **Business Data Catalog**. The BDC allows enterprise data to be exposed to Web Parts, InfoPath Forms Server, and search functions. Developers can use the BDC to build applications that allow users to interact with LOB data using familiar interfaces.

- **OpenXML file format**. Adoption of the OpenXML file format across the Office System applications facilitates rich server-side document manipulation.

Key Scenarios

SharePoint LOB applications are designed to interoperate using standard file formats and Web services:

- Open standards and interoperability are the core tenets of the Microsoft Office System.
- The metadata definitions of SharePoint LOB solution objects are based on Extensible Markup Language (XML) schemas.
- All Office products are service-enabled at all levels.
- Interoperable OpenXML file formats are the default schemas for business documents created by or generated for Microsoft Office business productivity client applications.

Design Considerations

While SharePoint provides many of the basic features you will use when interfacing with an LOB application, there are several key design issues that you must consider. These include user experience and the choice of client interface, as well as operational and maintenance issues.

Consider the following guidelines when designing a SharePoint LOB application:

- **Enable a user experience tailored to the user's role**. Provide different UI options based on the user's role. SharePoint contains functionality that allows you to automatically tailor the display based on user roles and groups. Utilize security groups or audience targeting to provide only the relevant options to users.
- **Integrate LOB systems with Office client applications**. Choose the patterns, such as the Direct Access pattern or Mediated pattern, to integrate LOB systems with Office client applications that are specific to the solution and the functional requirements. Consider ADO.NET or Web services for the Direct Access pattern. Consider using MOSS as a middle-tier application server for the Mediated pattern.
- **Avoid tight coupling between layers**. Use Web services to avoid tight coupling between the layers.
- **Consider remote data synchronization requirements**. All documents that are created, updated, or distributed should be synchronized with the LOB system and then stored for future use. Even though LOB systems are quite useful for handling transaction-oriented activities, they are not suited to capturing the significant work that occurs between activities.
- **Expose back-end LOB data through services for use in SharePoint and OBAs**. Exposing your back-end system via services allows SharePoint and OBA extensions to request, manipulate, and reformat data for the user. In this way, SharePoint can be used to extend back-end system behavior without extensive code development.

SharePoint LOB Frame

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

Table 1 SharePoint LOB frame

Category	Key issues
Documents and Content Storage	• Storing transient or transactional data in lists • Using the document library as source code control
Web Parts	• Not securing Web Parts • Deploying Web Parts in GAC when security is required • Combining multiple functionalities in a Web Part • Adding styles to controls in Web Parts
Workflow	• Failing to choose an appropriate workflow technology • Failing to consider workflow update scenarios
Business Data Catalog	• Overloading the staging area • Over-exposing data to users • No authentication while connecting to data sources
SharePoint Object Model	• Not releasing objects after use • Failing to choose an appropriate caching technology • Caching volatile data • Caching sensitive data
InfoPath Form Services	• External scripts accessing forms • Revealing sensitive information to the end user
Excel Services	• Data stores not authenticating the users • Not securing Open Data Connection files

Documents and Content Storage

Office documents, such as Excel spreadsheets, are stored in document libraries. You can use the Microsoft Office applications to consolidate diverse content from multiple data sources.

Consider the following guidelines when storing content in SharePoint:

- When storing documents in document libraries, use content types to define additional metadata of the document in a centralized way.
- Identify and plan your content types. Create the content type at the site level if it needs to be available on any child site. Create the content type at the list level if it needs to be available only to the list.
- Define unique metadata field names and their associations, document templates, and custom forms with the content types.
- Design content types to make use of their inheritance capabilities. Content types created at root sites can be automatically used in child sites. In addition, new content types can be derived and extended, starting from existing content types. Rather than manage each content type individually, use this behavior to simplify the maintenance of content types

- Consider customizing the Document Information Panel to collect content type metadata in order to track and edit metadata for documents. You can add business logic or data validation to the Document Information Panel.
- Consider storing user-configurable reference data or non-transient data in lists.
- Consider caching the contents of a list to a **DataTable** or **DataSet** if the list will be queried multiple times in your application.
- Do not treat SharePoint Lists as database tables. Use database tables for transient or transactional data.
- Do not replace file systems with SharePoint document libraries. Use a document library to store only the documents that require collaboration and management.
- Do not use SharePoint document libraries as source code control or as a platform for development team members to collaborate on source code. Use Microsoft Visual Studio® Team Foundation Server instead.
- Consider the restriction of a maximum of 2000 items per list container in document libraries and lists. Consider writing your own UI to retrieve items in lists when the list container exceeds 2000 items.
- Consider organizing documents into folders as opposed to using filtered views, for faster retrieval.

Web Parts

Web Parts allow you to provide rich composition at the presentation level. You can build custom Web Parts to implement application-specific functionality, and use Web Parts provided with SharePoint and other environments such as ASP.NET.

Consider the following guidelines when using Web Parts in your SharePoint LOB applications:

- Use Web Parts to interact with back-end LOB applications or Web services.
- Use Web Parts to allow users to create composite customizable interfaces that support personalization.
- Consider creating a custom code access security policy to increase the permissions of your Web Parts.

Consider the following guidelines when developing Web Parts:

- Identify suitable functionality that you would like to implement in Web Parts.
- Identify the data sources that the Web Parts will interact with (if any).
- Design Web Parts using layering guidelines to partition your presentation, business, and data logic in order to improve maintainability.
- Design Web Parts to perform only a single function in order to improve reuse.
- Design Web Parts to be configurable or customizable by users.
- Include a Web Part Manager in custom master pages that will be used by Web Part pages.
- Use Web Part zones to host Web Parts on Web Part Pages by users at run time.
- Consider using Web Part verbs to allow users to perform discrete actions.
- Avoid specifying style attributes directly on controls contained in Web Parts.
- Consider categorizing your properties to distinguish them from Web Part properties.

- Dispose properly of any SharePoint objects and unmanaged resources that you create in your Web Parts.

Workflow

SharePoint allows developers to create simple workflows and attach them to the document libraries in SharePoint. Users can also create custom workflows using the SharePoint Designer, or you can create custom workflows using Visual Studio.

Consider the following guidelines when designing workflows:

- Be clear on what business process or part of a business process is being automated. Consult a subject matter expert or business analyst to review existing business processes.
- Ensure that existing business processes are accurate and documented before implementing the workflows electronically.
- Choose the right workflow technology to meet the business requirements.
- Use out-of-the-box SharePoint workflows if business requirements are simple; for example, Approval.
- Consider using the SharePoint Designer to create workflows when out-of-the-box workflows cannot fulfill business requirements.
- Consider using Visual Studio to develop custom workflows when business requirements require complex workflows or integration with LOB systems.
- Consider using Visual Studio to create workflow activities that can be registered with SharePoint Designer in order to empower information workers.
- When developing custom workflows, choose the workflow type that is appropriate for your scenario. Consider state-based and sequential models.
- When debugging custom workflows, consider setting the logging level to verbose.
- When debugging custom workflows, consider implementing comprehensive instrumentation within your code.
- Consider versioning your workflow assemblies and changing the solution Globally Unique Identifier (GUID) when upgrading your old workflows.
- Consider the effect on existing workflow instances that are running when deploying newer versions.
- Consider creating separate workflow history lists and tasks list for workflows created by end users.
- Consider assigning workflows to content types in order to improve manageability. Assigning a workflow to a type implies that you can use a workflow in many different content libraries, but only have to maintain it in one place. Note that this functionality is available for out-of-the-box and Visual Studio workflows, but not for SharePoint Designer workflows.
- Consider that there can only be one running workflow instance of the same type per list item.
- Consider that workflow instances will only start on list items, and not the list itself.

Business Data Catalog

The Business Data Catalog (BDC) allows enterprise data to be exposed to Web Parts, InfoPath Forms Server, and search functions. Developers can use the BDC to build applications that allows users to interact with LOB data by using familiar interfaces.

Consider the following guidelines when developing BDC applications:

- Review the structure of data sources to ensure that they are suitable to be consumed directly by the BDC.
- Determine how the data will be used; for example, search, user profiles, or simple display.
- Consider using the Enterprise Single Sign-On features provided by SharePoint to authenticate to back-end data sources.
- Consider using the BDC Definition Editor from the Office Server SDK to minimize errors when creating the Application Definition File (ADF).
- Consider loading the BDCMedata.xsd schema into Visual Studio to minimize errors if you are manually editing the ADF.
- Consider using the most recent data access drivers for the data sources if possible in order to improve performance.
- Define an appropriate search scope to avoid over-exposing data.
- Consider using the BDC Security Trimmer for custom security trimming of entity instances, if required.

SharePoint Object Model

SharePoint exposes an object model that allows you to write code that automates processes. For example, you can implement custom versioning for documents, or enforce custom check-in policies.

Consider the following guidelines when writing custom code using the SharePoint object model:

- Dispose of the SharePoint objects that you have created after use to release unmanaged resources.
- Dispose of the SharePoint objects appropriately in exception handlers.
- Consider thread synchronization and thread safety when you must cache SharePoint objects.
- Consider loading the data from SharePoint objects into a **DataSet** or **DataTable** if caching is required.
- When elevating privileges, note that only new SharePoint objects created after elevation will use the elevated privileges.

InfoPath Form Services

InfoPath Form Services provides users with the capability to use browser-based forms based on form templates stored in SharePoint and exposed to the user through Microsoft Office InfoPath®. When deployed to a server running InfoPath Form Services, forms based on browser-compatible form templates (.xsn) can be opened in a Web browser from computers that do not have Office InfoPath 2007 installed, but they will open in Office InfoPath 2007 when it is installed.

Consider the following guidelines when designing to use InfoPath Forms for Form Services:

- Consider using Universal Data Connection (UDC) files for flexible management of data connections and reusability.
- Consider creating symmetrical forms—which look and operate exactly the same way whether they are displayed in the Office SharePoint Server Web interface, or within an Office system client application, such as Microsoft Office Word, Microsoft Office Excel, or Microsoft Office PowerPoint®.
- Use the Design Checker task pane of Office InfoPath to check for compatibility issues in browser forms.
- Consider selecting the **Enable browser-compatible features only** option when designing forms for the browser to hide unsupported controls.
- Consider submitting the form data to a database when reporting is required.
- Consider using multiple views, instead of a single view with hidden content, to improve the performance and responsiveness of your forms.
- Consider using Form View when configuring session state for InfoPath Forms Services.
- When exposing forms to public sites, ensure that form templates cannot be accessed by scripts or automated processes in order to prevent Denial of Service (DoS) attacks.
- When exposing forms to public sites, do not include sensitive information such as authentication information, or server and database names.
- Consider enabling protection to preserve the integrity of form templates and to prevent users from making changes to the form template.
- Do not use InfoPath Form Services when designing reporting solutions that require a large volume of data.
- Do not rely on the apparent security obtained by hiding information using views.
- Consider storing any sensitive information that is collected by the forms in a database.

Excel Services

Excel Services consists of three main components: Excel Calculation Services (ECS) loads the workbook, performs calculations, refreshes external data, and maintains sessions. Excel Web Access (EWA) is a Web Part that displays and enables interaction with the Excel workbook in a browser. Excel Web Services (EWS) is a Web service hosted in SharePoint that provides methods that developers can use to build custom applications based on the Excel workbook.

Consider the following guidelines when designing to use Excel Services:

- Consider configuring Kerberos authentication or single sign-on (SSO) for Excel Services to authenticate to SQL Server databases located on other servers.
- Publish only the information that is required.
- Configure the trusted file locations and trusted data connection libraries before publishing workbooks.
- Ensure that the Excel workbooks are saved to the trusted file locations before publishing.

- Ensure that Office Data Connection files are uploaded to the trusted data connection libraries before publishing workbooks.

Deployment Considerations

SharePoint LOB applications rely on SharePoint itself to provide much of the functionality. However, you must deploy the additional artifacts, such as components, in such a way that SharePoint can access and use them.

Consider the following guidelines when designing a deployment strategy for your SharePoint LOB applications:

- Determine the scope for your features, such as Farm, Web Application, Site Collection, or Site.
- Consider packaging your features into solutions.
- Consider deploying your assemblies to the BIN folder instead of the Global Assembly Cache in order to take advantage of the low-level Code Access Security mechanism.
- Test your solution after deployment using a non-administrator account.

Pattern Map

Category	Relevant patterns
Workflows	• Data-driven workflow • Human workflow • Sequential workflow • State-driven workflow

Pattern Descriptions

- **Data-driven workflow**. A workflow that contains tasks whose sequence is determined by the values of data in the workflow or the system.
- **Human workflow**. A workflow that involves tasks performed manually by humans.
- **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.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology for your SharePoint workflow, and provide guidance on creating Web Parts for custom SharePoint interfaces:

- If you require workflows that automatically support secure, reliable, transacted data exchange, a broad choice of transport and encoding options, and that 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 Microsoft BizTalk® Server.

- If you must interact with non-Microsoft systems, perform electronic data interchange (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 create ASP.NET Web Parts for your application, consider inheriting from the class *System.Web.UI.WebControls.WebParts.WebPart* unless you require backward compatibility with SharePoint 2003. If you must support SharePoint 2003, consider inheriting from the class *Microsoft.SharePoint.WebPartPages.WebPart*.

Additional Resources

For more information about using MOSS and WSS to build SharePoint LOB applications, see the following resources:

- *Developing Workflow Solutions with SharePoint Server 2007 and Windows Workflow Foundation* at http://msdn.microsoft.com/en-us/library/cc514224.aspx.
- *Best Practices: Common Coding Issues When Using the SharePoint Object Model* at http://msdn.microsoft.com/en-us/library/bb687949.aspx.
- *Best Practices: Using Disposable Windows SharePoint Services Objects* at http://msdn.microsoft.com/en-us/library/aa973248.aspx.
- *InfoPath Forms Services Best Practices* at http://technet.microsoft.com/en-us/library/cc261832.aspx.
- *White paper: Working with large lists in Office SharePoint Server 2007* at http://technet.microsoft.com/en-us/library/cc262813.aspx.

Cheat Sheet: Data Access Technology Matrix

Objectives

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

Overview

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

Data Access Technologies Summary

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

- **ADO.NET Core**. ADO.NET Core provides general retrieval, update, and management of data. ADO.NET includes providers for Microsoft® SQL Server®, OLE-DB, ODBC, SQL Server Compact Edition, and Oracle databases.
- **ADO.NET Data Services Framework**. This framework exposes data using the Entity Data Model, through RESTful Web services accessed over HTTP. The data can be addressed directly via a Uniform Resource Identifier (URI). The Web service can be configured to return the data as plain Atom and JavaScript Object Notation (JSON) formats.
- **ADO.NET Entity Framework**. This 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 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 that 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 business intelligence (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**. ADO.NET Sync Services is a provider included in the Microsoft Sync Framework synchronization for ADO.NET-enabled databases. It enables data synchronization to be built in occasionally connected applications. It periodically gathers information from the client database and synchronizes it with the server database.
- **Language-Integrated Query (LINQ)**. LINQ provides class libraries that extend C# and Microsoft Visual Basic® with native language syntax for queries. Queries can be performed against a variety of data formats, including DataSet (LINQ to DataSet), XML (LINQ to XML), in-memory objects (LINQ to Objects), ADO.NET Data Services (LINQ to Data Services), and relational data (LINQ to Entities).Understand that LINQ is primarily a query technology supported by different assemblies throughout the .NET Framework. For example, LINQ to Entities is included with the ADO.NET Entity Framework assemblies; LINQ to XML is included with the System.Xml assemblies; and LINQ to Objects is included with the .NET core system assemblies.
- **LINQ to SQL**. 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 Framework 4.0, LINQ to SQL scenarios will be integrated and supported by the ADO.NET Entity Framework; however, LINQ to SQL will continue to be a supported technology. For more information, see the ADO.NET team blog at 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. • Provides rich designer support in Microsoft 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 models. • Have 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 Entity Framework libraries need to be redeployed.

Technology	Benefits	Considerations
LINQ to Entities	• Is a LINQ-based solution for relational data in the ADO.NET Entity Framework. • Provides strongly typed LINQ access to relational data • Supports LINQ-based queries against objects built on top of the ADO.NET Entity Framework Entity Data Model. • Processing is on the server.	• Requires the ADO.NET Entity Framework
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 Framework 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 DataSet.	• Processing is all on the client side.
ADO.NET Sync Services	• Enables synchronization between databases, collaboration, and offline scenarios. • Synchronization can execute in the background. • Provides a hub-and-spoke type of architecture for collaboration between databases.	• Change tracking ability needs to be provided. • Exchanging large chunks of data during synchronization can reduce performance.

SOA / Service Scenarios

Technology	Benefits	Considerations
ADO.NET Data Services Framework	• Data can be addressed directly via a URI using a Representational State Transfer (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, Microsoft Silverlight®, and Asynchronous JavaScript and XML	• Is only applicable to service-oriented scenarios.

Technology	Benefits	Considerations
	(AJAX) client libraries allow developers to work directly with objects and provide strongly typed LINQ access to ADO.NET Data Services. • .NET, Silverlight, and AJAX client libraries provide a familiar API surface to Windows Azure Tables, SQL Data Services, and other Microsoft services.	
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, thereby reducing reusability. • The relational database structure may not match the object model, requiring you to write a data mapping layer by hand.
ADO.NET Data Services Framework	• Is a simple out-of-box solution with the ADO.NET Entity Framework. • Data can be addressed directly via a 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. • .NET, Silverlight, and AJAX client libraries provide a familiar API surface to Windows Azure Tables, SQL Data Services, and other Microsoft services.	• Is 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.

Technology	Benefits	Considerations
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 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	• 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 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 is much easier to use. • Query results can be used as parameters to **XElement** and **XAttribute** object constructors.	• Relies heavily on generic classes. • Is not optimized to work with untrusted XML documents, which require different mitigation techniques for security.

Technology	Benefits	Considerations
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 Framework 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 will evolve based on feedback received from the community.

General Recommendations

Consider the following 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 versus 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 the 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, available options include passing entity objects, Data Transfer Objects (DTO) that are mapped to entities, **DataSets**, 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 Windows Communication Foundation (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 want to store your data in the cloud, consider SQL Data Services.
- **Microsoft Windows Mobile®**. Many data technologies are too heavy for Windows Mobile devices, with their limited memory capabilities. Primarily utilize SQL Server Compact Edition and ADO.NET Sync Services to maintain data on a mobile device and synchronize it with a larger database system. Features such as merge replication can also assist in Windows Mobile scenarios.

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

Common Scenarios and Solutions

ADO.NET Core

Consider using ADO.NET Core if you:

- Need to use low-level APIs for full control over data access in your application.
- Want to leverage the existing investment in 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 a disconnected data-access experience.

ADO.NET Data Services Framework

Consider using the 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 an *N*-tier application and want to access data through a data-centric service interface.

ADO.NET Entity Framework

Consider using the ADO.NET Entity Framework 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.
- Are using Windows Mobile and want to sync with a central database server.

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 the 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 the LINQ syntax.

LINQ to XML

Consider using LINQ to XML if you:

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

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 on 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 at http://blogs.msdn.com/adonet/archive/2008/10/31/clarifying-the-message-on-l2s-futures.aspx

Mobile Considerations

A number of the technologies listed above are not available on the Windows Mobile operating system. The following technologies are not available on Windows Mobile at the time of publication:

- ADO.NET Entity Framework
- ADO.NET Data Services Framework
- LINQ to Entities

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

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

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*

Cheat Sheet: Integration Technology Matrix

Objectives

- Understand the tradeoffs for each integration technology choice.
- Understand the design impact of choosing integration technology.
- Understand all available integration technologies.
- Choose an integration technology for your scenario.

Overview

Use this cheat sheet to understand your technology choices for integration. Your choice of integration technology will be related to the application type you are developing. Use the Integration Technologies Summary to review each technology and its description. Use the Benefits and Considerations Matrix to make an informed choice of integration technology based on the advantages and considerations of each one. Use the Common Scenarios and Solutions to map your application scenario to common integration technology solutions.

Integration Technologies Summary

- **Microsoft® BizTalk® Server**. BizTalk provides a complete stack of adapters, orchestration, messaging, and protocols for building Enterprise Application Integration (EAI)–enabled systems.
- **Microsoft Host Integration Server**. Host Integration Server provides a platform for connecting applications with IBM zSeries and iSeries applications. In addition, Host Integration Server supports data connections between Microsoft Message Queuing (MSMQ) and IBM WebSphere MQ.
- **Microsoft Message Queuing (MSMQ)**. MSMQ allows you to connect applications using queued messaging. MSMQ provides guaranteed message delivery, priority-based messaging, and security. It can support integration with systems that may be occasionally connected or temporarily offline. MSMQ also supports both synchronous and asynchronous messaging scenarios.
- **Microsoft Enterprise Service Bus (ESB) Guidance**. ESB Guidance is a logical entity that provides a loosely coupled messaging architecture created using BizTalk Server. It exploits the underlying BizTalk features to provide flexible and extensible architecture. It provides capabilities such as delivery assurance, message security, service registry, intelligent routing, and unified exception handling.

Benefits and Considerations Matrix

The following table breaks down the benefits and considerations for each integration technology.

Table 1 Benefits and Considerations Matrix

Technology	Benefits	Considerations
BizTalk	• Enables electronic document exchange relationships between companies using Electronic Data Interchange (EDI) and/or Extensible Markup Language (XML) formats. • Integrates with non-Microsoft systems. • Easily extended to provide Enterprise Service Bus (ESB) capabilities. • WCF line-of-business (LOB) adapters enable development of custom adapters for use inside or outside BizTalk. • Provides out-of-box adapters for integration with systems such as SAP, Oracle, SQL, etc. • Provides SOAP adapter to help you to work with Web services.	• Might lead to tightly coupled infrastructure. • Requires customization to achieve ESB capabilities.
Host Integration Server	• Supports network integration between Microsoft Windows Server® and IBM mainframe or AS/400 computers. • Provides secure host access and identity management with support for Secure Sockets Layer (SSL)/Transport Layer Security (TLS), single sign-on (SSO), and password synchronization. • Provides data integration with support for MSMQ and XML-based Web services. • Includes a data access tool for creating and managing connections with Database 2 (DB2) databases. • Supports enterprise scalability and performance with simultaneous host sessions, load balancing, and hot failover. • BizTalk adapters for Host Systems are available to support BizTalk integration with DB2, IBM WebSphere MQ, Host Applications, and Host Files.	• Must be installed in a Windows Server environment. • Requires Microsoft Visual Studio® 2005 or greater. • Requires MSMQ with routing support.

Technology	Benefits	Considerations
Microsoft Message Queuing (MSMQ)	• Enables applications to communicate with each other across heterogeneous networks using message-based scenarios. • Supports reliable messaging between applications inside and outside of an enterprise. • Supports transactional capabilities, such as ensuring that messages are only delivered once, that messages are delivered in order, and confirmation that messages were retrieved from destination queues. • Provides message routing based on network topology, transport connectivity, and session concentration needs. • Allows message delivery over HTTP transport with support for SOAP Reliable Messaging Protocol (SRMP). • Supports the distribution of a single message to multiple destinations. • Is included with Windows Server 2003 and above. • Supports two deployment modes: domain mode with access to the Microsoft Active Directory® directory service, and Workgroup mode. • Includes WCF-provided endpoints for MSMQ.	• Deployment mode should be considered prior to installing and configuring MSMQ. • When using the Workgroup deployment mode, messages cannot be encrypted, internal certificates cannot be used, and cross-platform messaging is not supported. • Independent clients should be used instead of dependent clients. • MSMQ is optimized for sending remotely and receiving locally. As a result, you should avoid remote queue reads. • You should avoid functions that query Active Directory. • Asynchronous notifications using events can become lost. • WCF endpoints require Microsoft .NET Framework 3.0 or above.
Microsoft Enterprise Service Bus (ESB) Guidance	• Provides dynamic resolution of service endpoints at run time, which abstracts endpoint definition. • Decouples the message transformation from the application. • Integrates closely with WCF to provide security and reliable messaging. • Provides fault detection and reporting, through unified exception handling for both system and business exceptions. • Provides out-of-box resolvers for communicating with service registries such as Universal	• You need BizTalk server. • You might need customization for a specific business scenario. • Needs Dunda’s charts for exception management portal. • The EBS itinerary tracking data does not have out-of-box display.

Technology	Benefits	Considerations
	Description, Discovery and Integration (UDDI). • Supports an itinerary-based approach for routing and transformation. • Supports client-side and server itineraries. • Supports resolver extensibility for creating custom resolvers. • Supports designer for itinerary creation. • Supports exception management portal. • Exposes all key features such as exception handling, resolver, etc., as a Web service. • Provides itinerary tracking using Business Activity Monitoring (BAM).	

Common Scenarios and Solutions

BizTalk

Consider using BizTalk if you:

- Want interaction with multiple Web services via an orchestrator as part of a Service-Oriented Architecture (SOA).
- Want to support business-to-business (B2B) processes, including industry standards such as EDIFACT, ANSCI X12, HL7, HIPAA, or SWIFT.
- Want parallel execution of services.
- Need a solution that is highly reliable and requires a dedicated scalable server infrastructure with no code changes required.
- Need to measure business Key Performance Indicators (KPIs) by configuring a BAM solution to provide near real-time visibility into your application’s process data.
- Need to abstract your application business logic into declarative rule policies that can be easily changed in time to match dynamic changes of business requirements.

Host Integration Server

Consider using Host Integration Server if you:

- Need to support interaction with IBM zSeries or iSeries applications.
- Want to integrate BizTalk with DB2, WebSphere MQ, Host Applications, or Host Files.
- Want to integrate MSMQ with WebSphere MQ.

Microsoft Message Queuing

Consider using Microsoft Message Queuing if you:

- Need to support message-based interaction between applications.
- Want to integrate with non-Microsoft platforms.
- Need to support the SRMP.

Microsoft Enterprise Service Bus (ESB) Guidance

Consider using ESB Guidance if you:

- Need to support an itinerary-based approach.
- Need to support dynamic resolution and routing.
- Need to use dynamic transformations.
- Need to support robust and unified exception management for your EAI system.

Additional Resources

- For more information on BizTalk, see *BizTalk Server* at http://msdn.microsoft.com/en-us/biztalk/default.aspx
- For more information on Host Integration Server, see *Host Integration Server* at http://www.microsoft.com/hiserver/default.mspx
- For more information on MSMQ, see *Microsoft Message Queuing* at http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx
- For best practice information on MSMQ, see *MSMQ Best Practices* at http://download.microsoft.com/download/F/C/9/FC9989A2-DA75-4D96-B654-4BD29CF6AEE1/MSMQBestPractice.doc
- For more information see Microsoft ESB Guidance for BizTalk Server 2006 R2 at http://msdn.microsoft.com/en-us/library/cc487894.aspx

Cheat Sheet: patterns & practices Pattern Catalog

Composite Application Guidance for WPF

Category	Patterns
Modularity	• Service Locator
Testability Patterns	• Inversion of Control • Dependency Injection • Separated Presentation • Supervising Controller • Presentation Model

Data Patterns

Category	Patterns
Data Movement Patterns	• Data Replication • Master-Master Replication • Master-Subordinate Replication • Master-Master Row-Level Synchronization • Master-Subordinate Snapshot Replication • Capture Transaction Details • Master-Subordinate Transactional Incremental Replication • Master-Subordinate Cascading Replication
Pattlets	• Maintain Data Copies • Application-Managed Data Copies • Extract-Transform-Load (ETL) • Topologies for Data Copies

Enterprise Solution Patterns

Category	Patterns
Deployment Patterns	• Deployment Plan • Layered Application • Three-Layered Services Application • Tiered Distribution • Three-Tiered Distribution
Distributed Systems	• Broker • Data Transfer Object • Singleton

Category	Patterns
Performance and Reliability	• Server Clustering • Load-Balanced Cluster • Failover Cluster
Services Patterns	• Service Interface • Service Gateway
Web Presentation Patterns	• Model-View-Controller • Page Controller • Front Controller • Intercepting Filter • Page Cache • Observer
Pattlets	• Abstract Factory • Adapter • Application Controller • Application Server • Assembler • Bound Data Control • Bridge • Command(s) • Decorator • Façade • Four-Tiered Distribution • Gateway • Layer Supertype • Layers; Mapper • Mediator • MonoState • Observer • Naming Service • Page Data Caching • Page Fragment Caching • Presentation-Abstraction-Controller • Remote Façade • Server Farm • Special Case • Strategy • Table Data Gateway • Table Module • Template Method

Integration Patterns

Category	Patterns
Integration Layer	• Entity Aggregation • Process Integration • Portal Integration
Integration Topologies	• Message Broker • Message Bus • Publish / Subscribe
System Connections	• Data Integration • Functional Integration • Service-Oriented Integration • Presentation Integration
Additional Integration Patterns	• Pipes and Filters • Gateway

Web Services Security Patterns

Category	Pattern
Authentication	• Brokered Authentication • Brokered Authentication: Kerberos • Brokered Authentication: X509 PKI • Brokered Authentication: STS • Direct Authentication
Authorization	• Protocol Transition with Constrained Delegation • Trusted Subsystem
Exception Management	• Exception Shielding
Message Encryption	• Data Confidentiality
Message Replay Detection	• Message Replay Detection
Message Signing	• Data Origin Authentication
Message Validation	• Message Validator
Deployment	• Perimeter Service Router

Pattern Summaries

Composite Application Guidance for WPF

Modularity

- **Service Locator**. Create a service locator that contains references to the services and encapsulates the logic to locate them. In your classes, use the service locator to obtain service instances. See http://msdn.microsoft.com/en-us/library/cc707905.aspx

Testability

- **Dependency Injection**. Do not instantiate the dependencies explicitly in your class. Instead, declaratively express dependencies in your class definition. Use a **Builder** object to obtain valid instances of your object's dependencies and pass them to your object during the object's creation and/or initialization. See http://msdn.microsoft.com/en-us/library/cc707845.aspx
- **Inversion of Control**. Delegate the function of selecting a concrete implementation type for the classes' dependencies to an external component or source. See http://msdn.microsoft.com/en-us/library/cc707904.aspx
- **Presentation Model**. Separate the responsibilities for the visual display and the user interface (UI) state and behavior into different classes named, respectively, the view and the presentation model. The view class manages the controls on the UI, and the presentation model class acts as a façade on the model with UI-specific state and behavior, by encapsulating the access to the model and providing a public interface that is easy to consume from the view (for example, using data binding). See http://msdn.microsoft.com/en-us/library/cc707885.aspx
- **Separated Presentation**. Separate the presentation logic from the business logic into different artifacts. The Separated Presentation pattern can be implemented in multiple ways, such as **Supervising Controller** or **Presentation Model**, etc. See http://msdn.microsoft.com/en-us/library/cc707862.aspx
- **Supervising Controller**. Separate the responsibilities for the visual display and the event-handling behavior into different classes named, respectively, the view and the presenter. The view class manages the controls on the UI and forwards user events to a presenter class. The presenter contains the logic to respond to the events, update the model (business logic and data of the application), and, in turn, manipulate the state of the view. See http://msdn.microsoft.com/en-us/library/cc707873.aspx

Data Movement Patterns

- **Data Replication**. Build on the data movement building block by adding refinements that are appropriate to replication. See http://msdn.microsoft.com/en-us/library/ms978671.aspx .
- **Master-Master Replication**. Copy data from the source to the target and detect and resolve any update conflicts that have occurred since the last replication (due to changes to the same data on the source and target). The solution consists of a two replication links between the source and the target in opposite directions. Both replication links transmit the same replication set in both directions. Such a pair of replication links is referred to as *related links* in the more detailed patterns. See http://msdn.microsoft.com/en-us/library/ms978735.aspx
- **Master-Subordinate Replication**. Copy data from the source to the target without regard to updates that may have occurred to the replication set at the target since the last replication. See http://msdn.microsoft.com/en-us/library/ms978740.aspx

- **Master-Master Row-Level Synchronization**. Create a pair of related replication links between the source and target. Additionally, create a synchronization controller to manage the synchronization and connect the links. This solution describes the function of one of these replication links. The other replication link behaves the same way, but in the opposite direction. To synchronize more than two copies of the replication set, create the appropriate replication link pair for each additional copy. See http://msdn.microsoft.com/en-us/library/ms998434.aspx
- **Master-Subordinate Snapshot Replication**. Make a copy of the source replication set at a specific time (this is known as a snapshot), replicate it to the target, and overwrite the target data. In this way, any changes that might have occurred to the target replication set are replaced by the new source replication set. See http://msdn.microsoft.com/en-us/library/ms998430.aspx
- **Capture Transaction Details**. Create additional database objects, such as triggers and (shadow) tables, and record changes of all tables belonging to the replication set. See http://msdn.microsoft.com/en-us/library/ms978709.aspx
- **Master-Subordinate Transactional Incremental Replication**. Acquire the information about committed transactions from the source and replay the transactions in the correct sequence when they are written to the target. See http://msdn.microsoft.com/en-us/library/ms998441.aspx
- **Master-Subordinate Cascading Replication**. Increase the number of replication links between the source and target by adding one or more intermediary targets between the original source and the end target databases. These intermediaries are data stores that take a replication set from the source, and thus act as a target in a first replication link. They then act as sources to move the data to the next replication link and so on until they reach the cascade end targets (CETs). See http://msdn.microsoft.com/en-us/library/ms978712.aspx

Enterprise Solution Patterns

Deployment Patterns

- **Deployment Plan**. Create a deployment plan that describes which tier each of the application's components will be deployed to. While assigning components to tiers, if it is found that a tier is not a good match for a component, determine the cost and benefits of modifying the component to better work with the infrastructure, or of modifying the infrastructure to better suit the component. See http://msdn.microsoft.com/en-us/library/ms978676.aspx
- **Layered Application**. Separate the components of your solution into layers. The components in each layer should be cohesive and at roughly the same level of abstraction. Each layer should be loosely coupled to the layers underneath. See http://msdn.microsoft.com/en-us/library/ms978678.aspx
- **Three-Layered Services Application**. Base your layered architecture on three layers: the presentation, business, and data layers. This pattern presents an overview of the

responsibilities of each layer and the components that compose each layer. See http://msdn.microsoft.com/en-us/library/ms978689.aspx
- **Tiered Distribution**. Structure your servers and client computers into a set of physical tiers and distribute your application components appropriately to specific tiers. See http://msdn.microsoft.com/en-us/library/ms978701.aspx
- **Three-Tiered Distribution**. Structure your application around three physical tiers: the client, application, and database tiers. See http://msdn.microsoft.com/en-us/library/ms978694.aspx

Distributed Systems

- **Broker**. Use the Broker pattern to hide the implementation details of remote service invocation by encapsulating them into a layer other than the business component itself. See http://msdn.microsoft.com/en-us/library/ms978706.aspx
- **Data Transfer Object**. Create a data transfer object (DTO) that holds all data that is required for the remote call. Modify the remote method signature to accept the DTO as the single parameter and to return a single DTO parameter to the client. After the calling application receives the DTO and stores it as a local object, the application can make a series of individual procedure calls to the DTO without incurring the overhead of remote calls. See http://msdn.microsoft.com/en-us/library/ms978717.aspx
- **Singleton**. Singleton provides a global, single instance by making the class create a single instance of itself, allowing other objects to access this instance through a globally accessible class method that returns a reference to the instance. Additionally declare the class constructor as private so that no other object can create a new instance. See http://msdn.microsoft.com/en-us/library/ms998426.aspx

Performance and Reliability

- **Server Clustering**. Design your application infrastructure so that your servers appear to users and applications as virtual unified computing resources. One way to achieve this virtualization is by using a server cluster. A *server cluster* is the combination of two or more servers that are interconnected to appear as one, thus creating a virtual resource that enhances availability, scalability, or both. See http://msdn.microsoft.com/en-us/library/ms998414.aspx
- **Load-Balanced Cluster**. Install your service or application onto multiple servers that are configured to share the workload. This type of configuration is 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. See http://msdn.microsoft.com/en-us/library/ms978730.aspx
- **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*. 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. Each server in the cluster has at least one other server in the cluster identified as its standby server. See http://msdn.microsoft.com/en-us/library/ms978720.aspx

Services Patterns

- **Service Interface.** Design your application as a collection of software services, each with a service interface through which consumers of the application can interact with the service. See http://msdn.microsoft.com/en-us/library/ms998421.aspx
- **Service Gateway**. Encapsulate the code that implements the consumer portion of the contract into its own Service Gateway component. Service gateways play a similar role when accessing services as data access components do for access to the application's database. They act as proxies to other services, encapsulating the details of connecting to the source and performing any necessary translation. See http://msdn.microsoft.com/en-us/library/ms998420.aspx

Web Presentation Patterns

- **Model-View-Controller**. The Model-View-Controller (MVC) pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes. The Model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the View), and responds to instructions to change state (usually from the Controller). The View manages the display of information. The Controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate. See http://msdn.microsoft.com/en-us/library/ms978748.aspx
- **Page Controller**. Use the Page Controller pattern to accept input from the page request, invoke the requested actions on the model, and determine the correct view to use for the resulting page. Separate the dispatching logic from any view-related code. Where appropriate, create a common base class for all page controllers to avoid code duplication and increase consistency and testability. See http://msdn.microsoft.com/en-us/library/ms978764.aspx
- **Front Controller**. The Front Controller pattern solves the decentralization problem present in Page Controller by channeling all requests through a single controller. The controller itself is usually implemented in two parts: a handler and a hierarchy of commands. The handler receives the HTTP Post or Get request from the Web server and retrieves relevant parameters from the request. The handler uses the parameters from the request first to choose the correct command and then to transfer control to the command for processing. The commands themselves are also part of the controller. The commands represent the specific actions as described in the Command pattern. See http://msdn.microsoft.com/en-us/library/ms978723.aspx
- **Intercepting Filter**. Use the Intercepting Filter pattern to create a chain of composable filters to implement common pre-processing and post-processing tasks during a Web page request. See http://msdn.microsoft.com/en-us/library/ms978727.aspx

- **Page Cache**. Use a page cache for dynamic Web pages that are accessed frequently but change less often. See http://msdn.microsoft.com/en-us/library/ms978759.aspx
- **Observer**. Use the Observer pattern to maintain a list of interested dependents (observers) in a separate object (the subject). Have all individual observers implement a common Observer interface to eliminate direct dependencies between the subject and the dependent objects. See http://msdn.microsoft.com/en-us/library/ms978753.aspx

Integration Patterns

Integration Layer

- **Entity Aggregation**. Introduce an Entity Aggregation layer that provides a logical representation of the entities at an enterprise level, with physical connections that support the access and that update to their respective instances in back-end repositories. See http://msdn.microsoft.com/en-us/library/ms978573.aspx
- **Process Integration**. Define a business process model that describes the individual steps that make up the complex business function. Create a separate process manager component that can interpret multiple concurrent instances of this model and that can interact with the existing applications to perform the individual steps of the process. See http://msdn.microsoft.com/en-us/library/ms978592.aspx
- **Portal Integration**. Create a portal application that displays the information retrieved from multiple applications in a unified UI. The user can then perform the required tasks based on the information displayed in this portal. See http://msdn.microsoft.com/en-us/library/ms978585.aspx

Integration Topologies

- **Message Broker**. Extend the integration solution by using the Message Broker pattern. A *message broker* is a physical component that handles the communication between applications. Instead of communicating with each other, applications communicate only with the message broker. An application sends a message to the message broker, providing the logical name of the receivers. The message broker looks up applications registered under the logical name and then passes the message to them. See http://msdn.microsoft.com/en-us/library/ms978579.aspx
- **Message Bus**. Connect all applications through a logical component known as a message bus. A *message bus* specializes in transporting messages between applications. A message bus contains three key elements: a set of agreed-upon message schemas, a set of common command messages, and a shared infrastructure for sending bus messages to recipients. See http://msdn.microsoft.com/en-us/library/ms978583.aspx
- **Publish/Subscribe**. Extend the communication infrastructure by creating topics or by dynamically inspecting message content. Enable listening applications to subscribe to specific messages. Create a mechanism that sends messages to all interested subscribers. See http://msdn.microsoft.com/en-us/library/ms978603.aspx

System Connections

- **Data Integration**. Integrate applications at the logical data layer by allowing the data in one application (the source) to be accessed by other applications (the target). See http://msdn.microsoft.com/en-us/library/ms978572.aspx
- **Functional Integration**. Integrate applications at the business logic layer by allowing the business function in one application (the source) to be accessed by other applications (the target). See http://msdn.microsoft.com/en-us/library/ms978578.aspx
- **Service-Oriented Integration**. To integrate applications at the business logic layer, enable systems to consume and provide Extensible Markup Language (XML)-based Web services. Use Web Services Description Language (WSDL) contracts to describe the interfaces to these systems. Ensure interoperability by making your implementation compliant with the Web Services (WS-*) family of specifications. See http://msdn.microsoft.com/en-us/library/ms978594.aspx
- **Presentation Integration**. Access the application's functionality through the UI by simulating a user's input and by reading data from the screen display. See http://msdn.microsoft.com/en-us/library/ms978588.aspx

Web Service Services Security Patterns

Authentication

- **Brokered Authentication**. The Web service validates the credentials presented by the client, without the need for a direct relationship between the two parties. An authentication broker that both parties trust independently issues a security token to the client. The client can then present credentials, including the security token, to the Web service. See http://msdn2.microsoft.com/en-us/library/aa480560.aspx
- **Brokered Authentication: Kerberos**. Use the Kerberos protocol to broker authentication between clients and Web services. See http://msdn2.microsoft.com/en-us/library/aa480562.aspx
- **Brokered Authentication: X509 PKI**. Use brokered authentication with X.509 certificates issued by a certificate authority (CA) in a public key infrastructure (PKI) to verify the credentials presented by the requesting application. See http://msdn2.microsoft.com/en-us/library/aa480565.aspx
- **Brokered Authentication: STS**. Use brokered authentication with a security token issued by a Security Token Service (STS). The STS is trusted by both the client and the Web service to provide interoperable security tokens. See http://msdn2.microsoft.com/en-us/library/aa480563.aspx
- **Direct Authentication**. The Web service acts as an authentication service to validate credentials from the client. The credentials, which include proof-of-possession that is based on shared secrets, are verified against an identity store. See http://msdn.microsoft.com/en-us/library/aa480566.aspx

Authorization

- **Protocol Transition with Constrained Delegation**. Use the Kerberos protocol extensions in Microsoft® Windows Server®. The extensions require the user ID but not the password. You still need to establish trust between the client application and the Web service; however, the application is not required to store or send passwords. See http://msdn.microsoft.com/en-us/library/aa480585.aspx
- **Trusted Subsystem**. The Web service acts as a trusted subsystem to access additional resources. It uses its own credentials instead of the user's credentials to access the resource. See http://msdn2.microsoft.com/en-us/library/aa480587.aspx

Exception Management

- **Exception Shielding**. Sanitize unsafe exceptions by replacing them with exceptions that are safe by design. Return only those exceptions to the client that have been sanitized, or exceptions that are safe by design. Exceptions that are safe by design do not contain sensitive information in the exception message, and they do not contain a detailed stack trace, either of which might reveal sensitive information about the Web service's inner workings. See http://msdn2.microsoft.com/en-us/library/aa480591.aspx

Message Encryption

- **Data Confidentiality**. Use encryption to protect sensitive data that is contained in a message. Unencrypted data, which is known as *plaintext*, is converted to encrypted data, which is known as *cipher-text*. Data is encrypted with an algorithm and a cryptographic key. Cipher-text is then converted back to plaintext at its destination. See http://msdn.microsoft.com/en-us/library/aa480570.aspx

Message Replay Detection

- **Message Replay Detection**. Cache an identifier for incoming messages, and use message replay detection to identify and reject messages that match an entry in the replay detection cache. See http://msdn2.microsoft.com/en-us/library/aa480598.aspx

Message Signing

- **Data Origin Authentication**. Use data origin authentication, which enables the recipient to verify that messages have not been tampered with in transit (data integrity) and that they originate from the expected sender (authenticity). See http://msdn2.microsoft.com/en-us/library/aa480571.aspx

Message Validation

- **Message Validator**. The message validation logic enforces a well-defined policy that specifies which parts of a request message are required for the service to successfully process it. It validates the XML message payloads against an XML schema (XSD) to ensure that they are well-formed and consistent with what the service expects to process. The validation logic also measures the messages against certain criteria by examining the message size, the message content, and the character sets that are used. Any message that does not meet the criteria is rejected. See http://msdn2.microsoft.com/en-us/library/aa480600.aspx

Deployment

- **Perimeter Service Router**. Design a Web service intermediary that acts as a perimeter service router. The perimeter service router provides an external interface on the perimeter network for internal Web services. It accepts messages from external applications and routes them to the appropriate Web service on the private network. See http://msdn2.microsoft.com/en-us/library/aa480606.aspx

Additional Resources

- For information on patterns in the Composite Application Library, see *Composite Application Guidance for WPF* at http://msdn.microsoft.com/en-us/library/cc707841.aspx
- For information on data patterns, see *Data Patterns* at http://msdn2.microsoft.com/en-us/library/ms998446.aspx
- For information on enterprise solution patterns, see *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn2.microsoft.com/en-us/library/ms998469.aspx
- For information on integration patterns, see *Integration Patterns* at http://msdn2.microsoft.com/en-us/library/ms978729.aspx
- For information on Web Service Security, see *Web Service Security Guidance: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0* at http://msdn2.microsoft.com/en-us/library/aa480545.aspx

Cheat Sheet: Presentation Technology Matrix

Objectives

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

Overview

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

Presentation Technologies Summary

Mobile Applications

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

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

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

Rich Client Applications

The following presentation technologies are suitable for use in rich client applications:

- **Windows Forms**. This is the standard UI design technology for the .NET Framework. Even with the availability of Windows Presentation Foundation (WPF), Windows Forms is still a good choice for UI design if your team already has technical expertise with Windows Forms, and the application does not have any specific rich UI requirements.

- **Windows Forms with WPF user controls**. This technology allows you to take advantage of the more powerful UI capabilities provided by WPF controls. You can add WPF to your existing Windows Forms application. Keep in mind that WPF controls tend to work best on higher-powered client machines.
- **WPF application**. WPF applications support more advanced graphics capabilities, such as 2-D and 3-D graphics, display resolution independence, advanced document and typography support, animation with timelines, streaming audio and video, and vector-based graphics. WPF uses Extensible Application Markup Language (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**. This technology hosts a sandboxed WPF application in Microsoft Internet Explorer or Mozilla Firefox on Microsoft 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 Microsoft Windows Vista® or both .NET Framework 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**. This technology 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.

Rich Internet Applications (RIA)

The following presentation technologies are suitable for use in RIAs:

- **Silverlight**. This is a browser-optimized subset of WPF that works cross-platform and cross-browser. Compared to XBAP, Silverlight is a smaller, faster install but does not support 3-D graphics and text-flowable documents. Due to its small footprint and cross-platform support, Silverlight is a good choice for WPF applications that do not require premium WPF graphics support.
- **Silverlight with AJAX**. Silverlight natively supports Asynchronous JavaScript and XML (AJAX) and exposes its object model to JavaScript located in the Web page. You can use this capability to allow interaction between your page components and the Silverlight application.

Web Applications

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

- **ASP.NET Web Forms**. This is the standard UI design and implementation technology for .NET Web applications. An ASP.NET Web Forms application only needs to be installed on the Web server, with no components required on the client desktop.
- **ASP.NET Web Forms with AJAX**. Use AJAX with ASP.NET Web Forms to process requests between the server and client asynchronously to improve responsiveness, provide richer experience to the client, and reduce the number of postbacks to the server. AJAX is an integral part of ASP.NET in.NET Framework 3.5 and later.
- **ASP.NET Web Forms with Silverlight Controls**. If you have an existing ASP.NET application, you can use Silverlight controls to improve the user experience and avoid the requirement to write a whole new Silverlight application. This is a good approach for creating islands of Silverlight content in an existing application.
- **ASP.NET MVC**. This technology allows you to use ASP.NET to build applications based on the Model-View-Controller (MVC) pattern. ASP.NET MVC supports test-driven development and clear separation of concerns between UI processing and UI rendering. This approach helps to create clean HTML and avoid mixing presentation information with logic code.
- **ASP.NET Dynamic Data**. This technology allows you to create data-driven ASP.NET applications that leverage Language-Integrated Query (LINQ) to Entities functionality. It provides a rapid development model for line-of-business (LOB)-style data-driven applications, allowing simple scaffolding to full customization.

Benefits and Considerations Matrix

Mobile Applications

Technology	Benefits	Considerations
.NET Compact Framework	• Runs on the client machine for improved performance and responsiveness • Does not require 100-percent network connectivity. • Has a familiar programming model if you are used to Windows Forms. • Has Microsoft Visual Studio® Designer support. • Is usually installed in ROM on the device.	• Has a limited API compared to a desktop Windows Forms application. • Requires more client-side resources than an ASP.NET Mobile application. • Is not as easy to deploy over the Web as an ASP.NET Mobile application.
ASP.NET Mobile	• Supports a wide range of devices, including anything that has a Web browser. • Does not have a footprint on the device because no application must be installed. • Has a familiar programming model if you are used to ASP.NET Web Forms.	• Design support has been removed from Visual Studio 2008, but the controls will still render on devices. • Requires 100-percent network connectivity to run. • Performance and responsiveness are dependent on network bandwidth and latency. • Many devices now support full HTML

Technology	Benefits	Considerations
	• Templates can be downloaded off the Web for designer support with Visual Studio.	and ASP.NET, so ASP.NET Mobile may not be required.
Silverlight Mobile	• Offers rich UI and visualization, including 2-D graphics, vector graphics, and animation. • Silverlight code running on desktops can run on Silverlight Mobile. • Isolated storage is available to maintain objects outside the browser cache.	• Is an announced product in development, but not yet released at the time this document was published. • Uses more device resources than a Web application. • Desktop Silverlight applications running on mobile may require optimization to account for reduced memory and slower hardware. • Requires the Silverlight plug-in to be installed. • May not run on as many types of devices as Web applications because of plug-in install requirement.

Rich Client Applications

Technology	Benefits	Considerations
Windows Forms	• Has a familiar programming model. • Has Visual Studio Designer support. • Offers good performance on a wide range of client hardware.	• Does not support 3-D graphics, streaming media, flowable text, and other advanced UI features in WPF such as UI styling and templates. • 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, it may require higher-powered graphics hardware. • You cannot overlay Windows Forms and WPF controls, reducing visual design flexibility.
WPF application	• Provides rich UI and visualization including 2-D and 3-D graphics, display resolution independence, vector graphics, flowable text, and animation. • Supports variable-bandwidth streaming media (Adaptive Media Streaming). • XAML makes it easier to define the UI, data-binding, and events.	• Depending on the complexity of your UI, it 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.

Technology	Benefits	Considerations
	• Supports separate developer/graphic designer integration.	
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. • It 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. • Is easier to deploy and update than a WPF or Windows Forms application.	• Only works on Vista or on a client with .NET Framework 3.5 and the XBAP browser plug-in installed. • Only works in Internet Explorer and Mozilla Firefox browsers.

Rich Internet Applications (RIA)

Technology	Benefits	Considerations
Silverlight	• Provides a lightweight install for client machines. • Provides most of the UI and visualization power of WPF, such as media streaming, 2-D graphics, vector graphics, animation, and resolution independence. • Isolated storage provides an application cache independent from the browser cache. • Supports high-definition (HD) video. • Client-side processing provides improved user experience and responsiveness compared to a Web application. • Supports a wide variety of languages such as C#, VB.NET, Ruby, and	• Requires a Silverlight plug-in to be installed on the client. • Your team may be less familiar with Expression Blend compared to Visual Studio. • Lacks the advanced 3-D graphics and flowable text support of WPF. • Currently cannot make direct use of graphics acceleration on the client. • Is not easy to transition from WPF or XBAP due to differences in the XAML and controls.

Technology	Benefits	Considerations
	Python. • Supports windowless background processing as a replacement for JavaScript. • Provides cross-platform support, including Mac and Linux. • Provides cross-browser support, including Firefox and Safari.	
Silverlight with AJAX	• Allows you to use existing AJAX libraries and routines from your Silverlight application. • Allows Silverlight objects to be dynamically created and destroyed through communication with the server as the user interacts with the application.	• May be an unfamiliar programming model if your team is used to pure ASP.NET or Silverlight.

Web Application

Technology	Benefits	Considerations
ASP.NET Web Forms	• Brings a development experience similar to Windows Forms to the Web. • Has no client dependency. • Requires no installation on the client. • Provides cross-platform and cross-browser support. • Provides Visual Studio design support. • Makes a large number of controls available.	• UI is limited to HTML and Dynamic HTML (DHTML) support. • Client-side storage is limited to cookies and view-state. • Updating page contents requires a full postback and page refresh. • Has limited UI responsiveness because all processing occurs on the server.
ASP.NET Web Forms with AJAX	• Provides a richer look and feel than a traditional Web Forms application. • Provides improved UI responsiveness and a richer experience. • Supports lazy loading. • Allows partial page refreshes. • An integral part of ASP.NET 3.5.	• May be an unfamiliar programming model if your team is used to pure ASP.NET. • Does not work if JavaScript is disabled on the client.
ASP.NET Web Forms with Silverlight Controls	• Allows you to add Silverlight rich visualization and UI to existing ASP.NET applications. • Provides a strategy for transition to	• Requires the Silverlight plug-in to be installed on the client. • Your team may be less familiar with Expression Blend compared

Technology	Benefits	Considerations
	full Silverlight applications.	to Visual Studio.
ASP.NET MVC	• Supports test-driven development. • Enforces separation between UI processing and UI rendering. • Allows you to create user-friendly and search engine–friendly URLs. • Provides full control over markup. • Provides full control over how content is rendered. • Navigation is controlled by configuration to greatly reduce the amount of code required.	• Does not support View state. • No support for control events.
ASP.NET Dynamic Data	• Allows the creation of fully data-driven sites that render automatically. • Has built-in support for LINQ querying languages. • Has built-in support for the ADO.NET Entity Framework. • LINQ allows you to model your database to create object-to-data mappings.	• Currently there are only a few controls that support the technology.

Common Scenarios and Solutions

Mobile Applications

.NET Compact Framework

Consider using the .NET Compact Framework if:

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

ASP.NET for Mobile

Consider using ASP.NET for Mobile if:

- Your team has ASP.NET expertise and you want to target a wide range of mobile devices.
- You are building an application that must have no client-side installation or plug-in dependencies.

- You are building an application that can rely on 100-percent network connectivity.
- You want to support the widest possible range of devices.
- You want to use as few device resources as possible.

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

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 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 its rich visualization and UI capabilities.
- You want to use XAML to define your UI design, data binding, and events.

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 the 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 Framework 3.5 with the XBAP browser plug-in installed.

Rich Internet Applications

Silverlight

Consider using Silverlight if:

- You are building an Internet-facing Web application or an internal application that requires seamless deployment.
- You want to leverage the rich visualization and UI capabilities of Silverlight.
- You want to leverage the streaming-media capabilities of Silverlight.
- You want a browser-optimized subset of WPF and XAML; or, more generally, a subset of .NET.
- You are targeting cross-platform and cross-browser clients.

Silverlight with AJAX

Consider using Silverlight with AJAX if:

- You want direct access to the Silverlight object model from your Web page.
- You want to manipulate Silverlight controls based on user interaction with your Web page.

Web Applications

ASP.NET Web Forms

Consider using ASP.NET Web Forms if:

- Your team already has experience building ASP.NET Web Forms.
- You have an existing ASP.NET Web Forms application that you want to extend or modify.
- You want to run on the widest possible range of client machines.
- You do not want to install anything on the client.
- You want to design simple functionality such as Create-Read-Update-Delete (CRUD) operations without a rich UI or animation.

ASP. NET Web Forms with AJAX

Consider using ASP. NET Web Forms with AJAX if:

- You want to create ASP.NET Web Forms with a more responsive and richer user experience.
- You want to support lazy loading and partial page refreshes.

ASP. NET Web Forms with Silverlight Controls

Consider using ASP. NET Web Forms with Silverlight Controls if:

- You already have an ASP.NET Web Forms application and want to leverage the rich visualization and UI capabilities of Silverlight.

- You are planning to transition your Web application to Silverlight.

ASP.NET MVC
Consider using ASP. NET MVC if:
- You want to implement the Model-View-Controller (MVC) pattern.
- You want full control over your markup.
- You want to implement a clear separation of concerns between UI processing and UI rendering.
- You want to follow test-driven development.

ASP.NET Dynamic Data
Consider using ASP.NET Dynamic Data if:
- You want to rapidly build a data-driven application.
- You want to use the LINQ query language or the Entity Framework data model.
- You want to use the built-in modeling capabilities of LINQ to more easily map your objects to data.

Additional Resources

- For information on Silverlight, see the official Silverlight Web site at http://silverlight.net/default.aspx
- For information on "Islands of Richness," see Islands of Richness with Silverlight on an ASP.NET page at http://blogs.msdn.com/brada/archive/2008/02/18/islands-of-richness-with-silverlight-on-an-asp-net-page.aspx

Cheat Sheet: Workflow Technology Matrix

Objectives

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

Overview

Use this cheat sheet to understand your technology choices for designing workflows. Your choice of workflow technology will be related to the type of workflow you are developing. Use the Workflow Technologies Summary to review each technology and its description. Use the Benefits and Considerations Matrix to make an informed choice of workflow technology based on the advantages and considerations of each one. Use the Common Scenarios and Solutions to map your application scenario to common workflow technology solutions.

Workflow Technologies Summary

- **Windows Workflow Foundation (WF)**. WF is a foundational technology that allows you to implement workflow. A toolkit for professional developers and independent software vendors (ISVs) who want to build a sequential or state-machine based workflow, WF supports the following types of workflow: Sequential, State-Machine, Data Driven, and Custom. You can create workflows using the Windows Workflow Designer in Microsoft® Visual Studio®.
- **Workflow Services**. Workflow Services provides integration between Windows Communication Foundation (WCF) and Windows Workflow Foundation (WF) to provide WCF-based services for workflow. Starting with Microsoft .NET Framework 3.5, WCF has been extended to provide support for workflows exposed as services and the ability to call services from within workflows. In addition, Microsoft Visual Studio 2008 includes new templates and tools that support workflow services.
- **Microsoft Office SharePoint® Services (MOSS)**. MOSS is a content-management and collaboration platform that provides workflow support based on WF. MOSS provides a solution for human workflow and collaboration in the context of a SharePoint server. You can create workflows for document approval directly within the MOSS interface. You can also create workflows using either the Microsoft Office SharePoint Designer or the Windows Workflow Designer in Visual Studio**.** For workflow customization, you can use the WF object model within Visual Studio.
- **Microsoft BizTalk® Server**. BizTalk currently has its own workflow engine that is geared toward orchestration, such as enterprise integration with system-level workflows. A future version of BizTalk may use WF as well as XLANG, the existing orchestration technology in

BizTalk. You can define the overall design and flow of loosely coupled, long-running business processes by using BizTalk Orchestration Services within and between applications.

Human Workflow vs. System Workflow

The term *workflow* applies to two fundamental types of process:

- **Human workflow**. *Human workflow* is a type of workflow in which a process that includes human collaboration is decomposed into a series of steps or events. These events flow from one step to the next based on conditional evaluation. The majority of the time, workflow is composed of activities that are carried out by humans.
- **System workflow**. *System workflow*, or orchestration, is a specific type of workflow that is generally applied to implement mediation between business services from business processes. Orchestration does not include human-performed activities.

Benefits and Considerations Matrix

The following table breaks down the benefits and key considerations for each of the workflow technologies.

Table 1 Benefits Consideration Matrix

Technology	Benefits	Considerations
Windows Workflow Foundation (WF)	• A developer-centric solution for creating workflows. • Supports sequential, state-machine, and data-driven workflows. • Designer support available in Visual Studio. • Includes protocol facilities for secure, reliable, transacted data exchange. • Supports long-running workflows that can persist across system restarts.	• Custom code is required if you want to host the designer in your application. • Does not provide true parallel execution support.
Workflow Services	• Provides integration between WCF and WF. • Allows you to expose workflows to client applications as services. • Supports coordination across multiple services to complete a business process. • When calling Workflow Services, the WF run time is automatically engaged for new or existing workflow instances. • Provides developer support in	• Requires .NET Framework 3.5 or greater. • Extra coding is required when not using default security credentials.

Technology	Benefits	Considerations
	Visual Studio 2008, with new templates and tools for Workflow Services.	
MOSS 2007 Workflow	• Workflow engine is based on WF. • Approval-based workflow can be defined by using the Web interface. • SharePoint Designer can be used to define conditional or data-driven workflows. • Visual Studio can be used to create custom workflows using WF components and services. • Integrates with applications in the Microsoft Office suite.	• Workflows are bound to a single site, and cannot access information in other sites. • Is not well suited for complex line-of-business (LOB) integrated workflow solutions.
BizTalk	• Provides a single solution for business process management. • Enables electronic document exchange relationships between companies using Electronic Data Interchange (EDI) and/or Extensible Markup Language (XML) formats. • Contains orchestration capabilities for designing and executing long-running, loosely coupled business transactions. • Integrates with non-Microsoft systems. • Easily extended to provide Enterprise Service Bus (ESB) capabilities. • WCF LOB adapters enable development of custom adapters for use inside or outside BizTalk.	• Saves the orchestration state to the Microsoft SQL Server® and causes latency while executing the orchestration. • Current version does not use WF. However, a future version may support WF.

Common Scenarios and Solutions

Windows Workflow Foundation

Consider using WF if you:

- Want to build a custom workflow solution.
- Want workflow designer support in Visual Studio.
- Want to host the WF designer in your application.

Workflow Services

Consider using Workflow Services if you:

- Want to expose workflows as services.
- Want to call services from within a workflow.
- Want to coordinate calls across multiple services to complete a business process.

MOSS 2007 Workflow

If you are already using SharePoint, consider using MOSS 2007 workflow if you:

- Want to enable workflow for human collaboration.
- Want to enable workflow on a SharePoint list or library; for example, to support an approval process.
- Want to extend SharePoint workflow to add custom tasks.
- Want to use the workflow designer in Visual Studio.

BizTalk

Consider using BizTalk if you:

- Are looking for a workflow solution that works across different applications and systems.
- Want a server-hosted system workflow product that enables enterprise integration.
- Are developing an application that must gather data from multiple Web services as part of a Service-Oriented Architecture (SOA).
- Are developing an application that has long-running business processes that may take many days to complete.
- Want to support business to business (B2B) processes based on industry standards.
- Want parallel execution of services.
- Need to abstract your application business logic into declarative rules that can be easily changed to match changing business requirements.

Additional Resources

- For more information on MOSS 2007 workflows, see *Workflows in Office SharePoint Server 2007* at http://msdn.microsoft.com/en-us/library/ms549489.aspx.
- For more information on WF, see *Windows Workflow Foundation* at http://msdn.microsoft.com/en-us/netframework/aa663328.aspx.
- For more information on Workflow Services, see *Workflow Services* at http://msdn.microsoft.com/en-us/library/cc825354.aspx
- For more information on BizTalk, see *BizTalk Server* at http://msdn.microsoft.com/en-us/biztalk/default.aspx
- For more information on enterprise workflows, see *Architecting Enterprise Loan Workflows and Orchestrations* at http://msdn.microsoft.com/en-us/library/bb330937.aspx