Microsoft

Service Architecture Guide

Application Architecture Pocket Guide Series

patterns & practices

Service Architecture Guide

patterns & practices

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

Introduction

Overview

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

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

- **Chapter 1, "Service Architecture,"** provides general design guidelines for a service application, explains the key attributes, discusses the use of layers, provides guidelines for performance, security, and deployment, and lists the key patterns and technology considerations.
- **Chapter 2, "Designing Services,"** helps you to understand the key scenarios for designing services including scenarios for both service providers and service consumers. Use these scenarios to help you better understand how to design your service architecture.
- **Chapter 3, "Service Layer Guidelines,"** helps you to understand how the service layer fits into the typical application architecture, learn about the components of the service layer, learn how to design these components, and understand common issues faced when designing a service layer. It also contains key guidelines for designing a service layer, and lists the key patterns and technology considerations.
- **Chapter 4, "Communication Guidelines,"** helps you to learn the guidelines for designing a communication approach, and understand the ways in which components communicate with each other. It will also help you to learn the interoperability, performance, and security considerations for choosing a communication approach, and the communication technology choices available.

Why We Wrote This Guide

We wrote this guide to accomplish the following:

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

Features of This Guide

- **Framework for application architecture**. The guide provides a framework that helps you to think about your application architecture approaches and practices.

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

Audience

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

Ways to Use the Guide

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

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

Feedback and Support

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

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

Technical Support

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

Community and Newsgroup Support

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

The Team Who Brought You This Guide

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

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

Contributors and Reviewers

- **Test Team**. Rohit Sharma; Praveen Rangarajan
- **Edit Team**. Dennis Rea

Tell Us About Your Success

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

Chapter 1 – Service Architecture

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.

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 them. 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 WSDL file, and operations are called using 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.

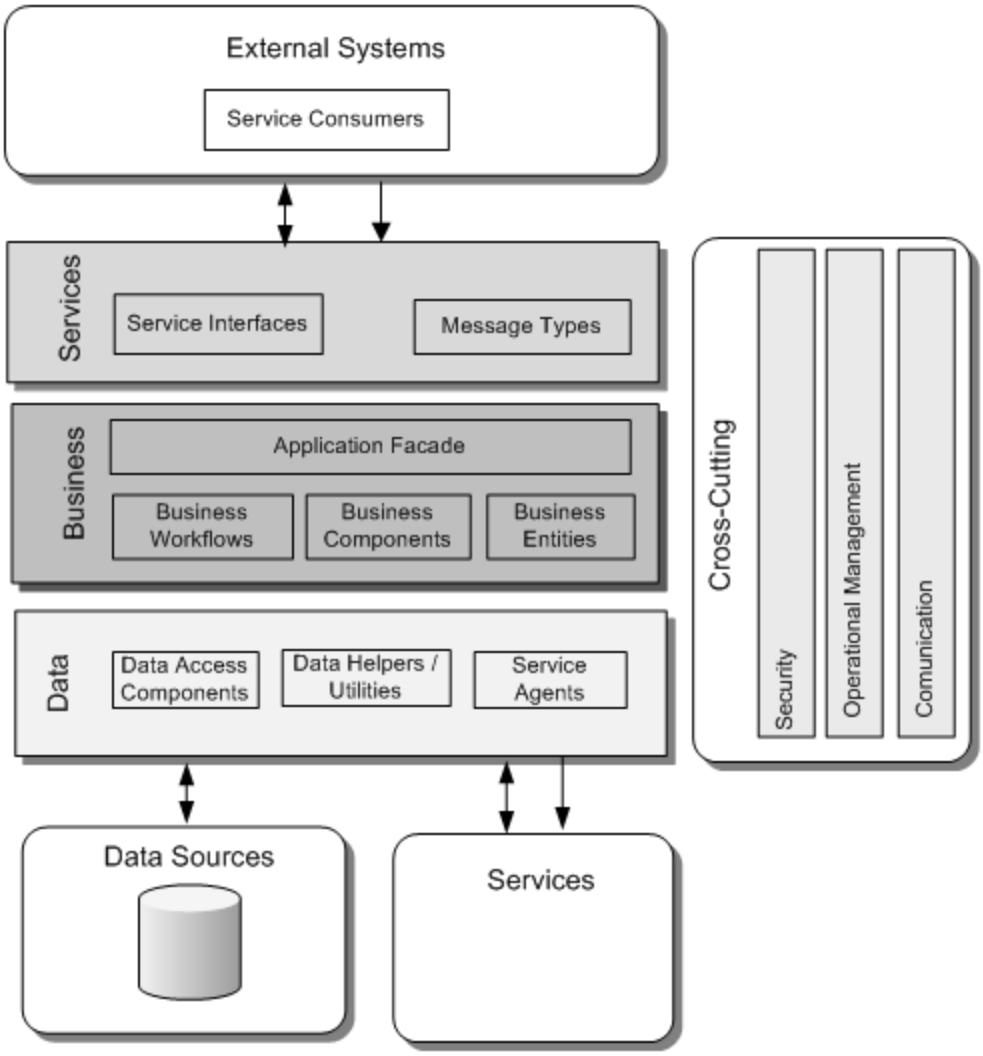

Figure 1 - Common services application architecture.

Common Services Scenarios

Services are, by nature, flexible, 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 by Web applications or Smart Client applications over the Internet. Decisions on authentication and authorization must be made based upon 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 the Intranet**. This scenario describes a service that is consumed by Web applications or Smart Client applications over an intranet. Decisions on authentication and authorization must be made based upon intranet trust boundaries and credentials

options. For example, Active Directory 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 made based upon 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. For example, 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, honor the service contract, and anticipate invalid requests or invalid request order. 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 Application (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 may be designing an Operational Data Store (ODS) that provides a service based interface.

General

- **Design coarse-grained operations**. Avoid chatty calls to the service, 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 may arrive out of order, implement a design that will store messages and then process them in the correct order.

SOA Services

- **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 sub-sets 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 that external consumers interact with. 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

- **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

- **Use interfaces supported by your workflow engine.** Attempting to create custom interfaces can restrict the type 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.
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	• Not catching exceptions that can be handled. • Not logging exceptions. • Compromising message integrity when an exception occurs.
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	• Providing limited schema support. • Not appreciating that tools for REST are primitive. • Using hypertext to manage state.
SOAP	• Not choosing the appropriate security model. • Not planning for fault conditions. • Using complex types in the message schema.
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. If the service is hosted in IIS, you can take advantage of the

authentication support provided by IIS. If the service is hosted using a Windows Service or a console application, you must use message-based authentication.

When designing an authentication strategy, consider following guidelines:

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

Authorization

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

When designing an authorization strategy, consider following guidelines:

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

Communication

When designing the communication strategy for your service 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 TCP for more efficient communication. If the service will be deployed in to a public facing network, you should choose the HTTP protocol.

When designing a communication strategy, consider following guidelines:

- Determine how to reliability 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. Failing 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, and it is important that the design take into account the impact on performance. A good approach is to design a centralized

exception management and logging mechanism, and consider providing access points that support instrumentation and centralized monitoring in order to assist system administrators.

When designing an exception management strategy, consider following guidelines:

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

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 type of operations you need to support. For example, you may be exchanging documents, executing commands, or raising events. When using slow message delivery channels, you should also consider including expiration information in the message.

When designing a message construction strategy, consider following guidelines:

- Determine the appropriate patterns for message constructions (such as Command, Document, Event, and Request-Reply).
- Divide very large quantities of data into 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.

When designing message endpoints, consider following guidelines:

- Determine relevant patterns for message endpoints (such as Gateway, Mapper, Competing Consumers, and Message Dispatcher).
- Determine if you should accept all messages, or implement a filter to handle specific messages.
- Design for idempotency in your message interface. Idempotency is the situation where you could receive duplicate messages from the same consumer, but should only handle one. In other words, an idempotent endpoint will guarantee that only one message will be handled, and all duplicate messages will be ignored.

- Design for commutativity. 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 may need to support guaranteed delivery.

Message Protection

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

When designing message protection, consider following guidelines:

- If your messages will not be routed through multiple servers, consider using transport layer security such as SSL.
- If the message passes through one or more servers, always 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.
- Use 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.

When designing message transformation, consider following guidelines:

- 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.

When designing message exchange patterns, consider the following guidelines:

- 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 non-standard contracts.

Representational State Transfer (REST)

REpresentational State Transfer (REST) is an architecture style for distributed systems designed to reduce complexity by dividing a system into resources. A Uniform Resource Indicator (URI) identifies each resource. The operations supported by a resource represent the functionality provided by a service that uses REST. The URI used to identify a REST resource is normally a Uniform Resource Locator (URL), which represents the Web address for the resource.

When designing REST resources, consider following guidelines:

- Identify and categorize resources that will be available to clients.
- Choose an approach for resource identification. A good practice would be to use meaningful names for REST starting points and unique identifiers, such as a GUID, for specific resource instances.
- 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.

SOAP

Simple Object Access Protocol (SOAP) is a message-based protocol used to implement the message layer of a service. The message is composed of an envelope that contains a header and body. The header can 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.

When designing SOAP messages, consider following guidelines:

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

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.

When designing a validation strategy, consider following guidelines:

- 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 may 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.

When designing your service layer, consider the following guidelines:

- Do not implement business rules in the services layer. The service layer should only be responsible for managing service requests and 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 entity value objects, or lightweight domain objects. The main issue is that you should not include business rules in these business entities.

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.

When designing you business layer, consider the following guidelines:

- 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 term "façade" is French for “front” or “face”, and is often used to mean a "false front". In software, a façade represents the main entry point into a 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. A service can handle a large number of consumer interactions, and 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, SharePoint lists, or any other applications that manage data.

When designing your data layer, consider the following guidelines:

- 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, it is vital to use operations that are as coarse-grained as possible. This depends on the type of service you are designing. In an SOA service, the

operations should be application-scoped. For example, you should define an operation that returns all of the demographic data for a person in one call; you should not define operations that return portions of demographic data because this would require multiple calls to retrieve the data.

Never think of services as being components with fine-grained operations that provide building blocks for an application. A service is an application that provides operations that are used to support different 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. The result would be poor performance, coupled with a design that is hard to extend and maintain.

When designing for maximum performance, consider the following guidelines:

- 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.

When designing a security strategy, consider the following guidelines:

- 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.

When deploying a services application, consider following guidelines:

- 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
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 the request/reply pattern.
- **Durable Subscriber -** In a disconnected scenario, messages are saved and then made accessible to the client when connecting to the message channel to provide guaranteed delivery.
- **Entity -** (REST) Resources that can be read with a GET operation, but can only be changed by PUT and DELETE operations.
- **Envelope Wrapper -** A wrapper for messages that contains header information used, for example, to protect, route, or authenticate a message.
- **Event Message -** A structure that provides reliable asynchronous event notification between applications.
- **Exception Shielding -** Prevent a service from exposing information about its internal implementation when an exception occurs.
- **Façade** – Implement a unified interface to a set of operations to provide a simplified reduce coupling between systems.
- **Fault Contracts -** A schema that defines errors or faults that can be returned from a service request.
- **Federation -** An integrated view of information distributed across multiple services and consumers.
- **Fire and Forget -** A one-way message communication mechanism used when no response is expected.
- **Idempotent Receiver -** Ensure that a service will only handle a message once.
- **Long running 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 PUT.
- **Transaction -** (REST) Resources that support transactional operations.
- **Transactional Client -** A client that can implement transactions when interacting with a service.
- **Validation -** Check the content and values in messages to protect a service from malformed or malicious content.

Technology Considerations

The following 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-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 2 - Designing Services

Objectives

- Learn the architecture for services.
- Understand the key scenarios for service providers and service consumers.
- Understand the design considerations for services.
- Learn about scenarios related to services in the cloud.

Overview

A service is a program or an application with which consumers can interact by exchanging well-defined messages. 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. From a design perspective, you must consider two different scenarios for services and applications: as a provider of services, and as a consumer of services. In many cases, you must consider both perspectives when designing an application that provides and consumes services.

This chapter starts with an overview of service architecture, followed by key scenarios related to both providing and consuming services. Following the overview and scenarios, you will learn about design guidelines related to providing and consuming services. Finally, you will learn about the evolution of services into the cloud, where they can be combined with client-based software to provide application or platform services.

Service Architecture

Figure 1 shows a typical service application architecture that conforms to the commonly used layered design approach.

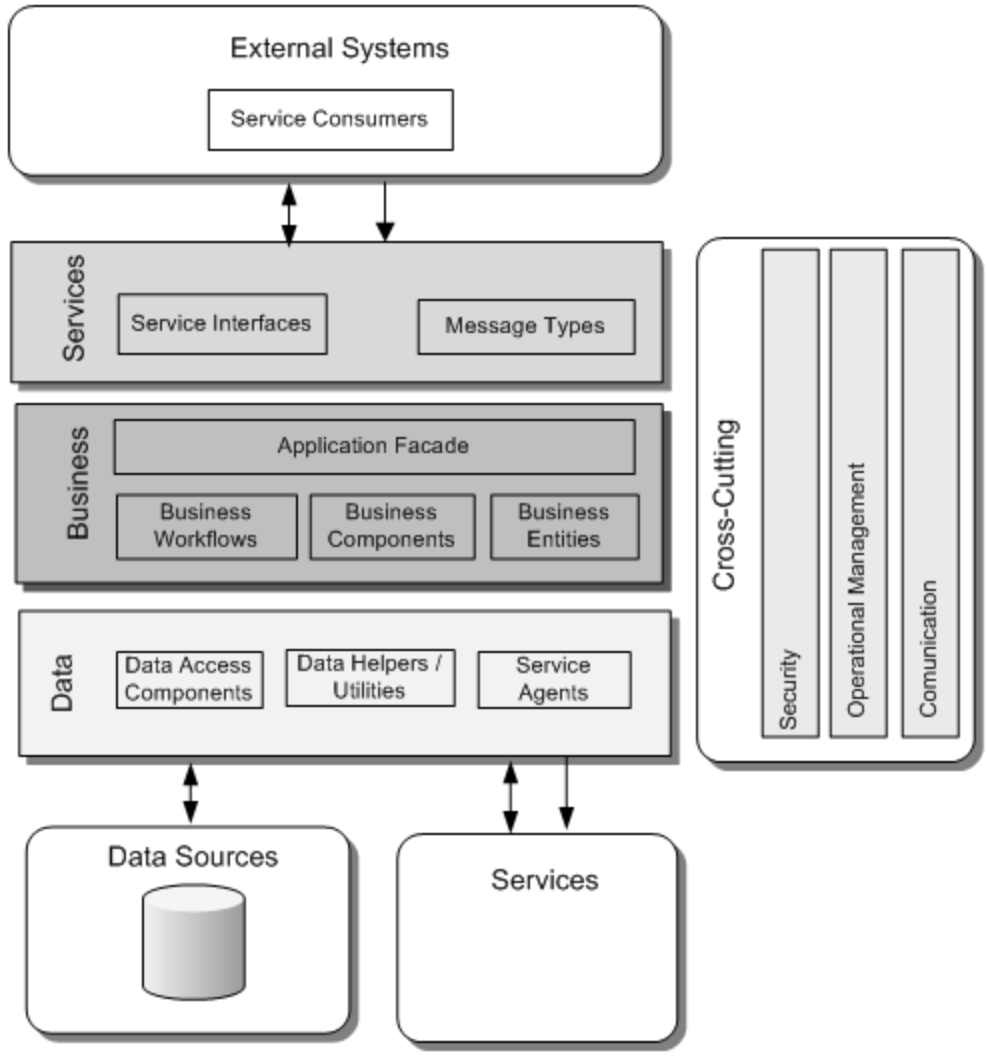

Figure 1 - A typical service application architecture.

The service architecture comprises the following elements:

- **Services layer**. This layer defines the operations provided by the service interface, and the messages required to interact with each operation. The service layer is described by a service contract, which specifies the service behavior and the messages required as the basis for interaction. The service layer contains message types that wrap data contracts used to support service operations.
- **Business layer**. This layer incorporates components that implement the business logic of the service. The business layer also includes business entities that represent objects specific to the application domain. These entities can be data structures or full business objects with data and behavior. This layer may also include business workflows. These workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools such as Windows Workflow Foundation (WF).
- **Data layer**. This layer contains the logic necessary to access data. It also contains service agents, which isolate the idiosyncrasies of calling diverse services from your application. The

data layer can also include helpers and utilities to support different features related to data access, such as a basic mapping between the format of the data from data sources or services and the format your application requires.

- **Components to address cross cutting concerns**. These components address functionality common to multiple layers of the architecture, such as exception management and instrumentation.

Key Scenarios

When it comes to services your application can be a provider that exposes services, a consumer that uses services, or a combination where services are exposed and used by the application. As a result, the key scenarios can be grouped into two categories; service provider and service consumer. Figure 2 shows a common deployment scenario for a Rich Client application. In this example, the code in the application layer represents a service provider that is exposing a service for use by the client. The code on the client represents consumers of the service.

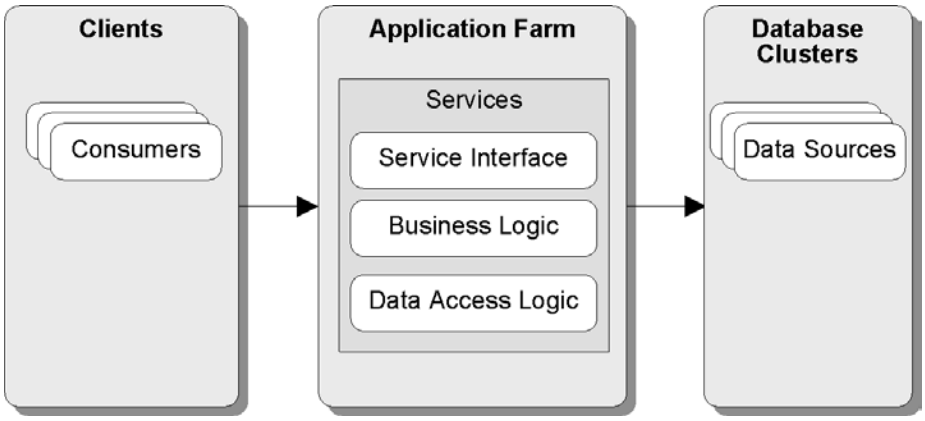

Figure 2 - A common Rich Client deployment scenario where the application layer is a service provider.

Service Provider Scenarios

Services are, by nature, flexible, and can be used in a wide variety of scenarios and combinations. The following are key typical scenarios used when providing services:

- **Service exposed over the Internet.** This scenario describes a service that is consumed by Web applications or Smart Client applications over the Internet. Decisions on authentication and authorization must be made based upon 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 the Intranet**. This scenario describes a service that is consumed by Web applications or Smart Client applications over an intranet. Decisions on authentication and authorization must be made based upon intranet trust boundaries and credentials

options. For example, Active Directory 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 made based upon 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. For example, 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.

Service Consumer Scenarios

Many applications use services to provide an interface between the presentation tier and application tier of a distributed design or between the presentation layer and business layer of a web design. In addition, services can be used to provide a data access interface for enterprise solutions that use a common data model. Many organizations and product companies also provide services that are used to extend or supplement the functionality of an application.

- **Presentation components accessing application services.** Web applications often use a message-based interface between the presentation layer and business layer of a design. In this scenario, operations from the business layer are exposed as services consumed by presentation layer components. With distributed applications, where presentation components are deployed to a separate tier, services are often used to provide the interface between presentation components and the application tier.
- **Application tier accessing data services.** This scenario is common in a large enterprise that uses an Operational Data Store (ODS) as a common data source. Instead of using traditional data access methods, an application tier would use services to interact with the data store to perform data centric operations.
- **Application using external services.** Many applications depend on functionality that is provided by external services. These services could be provided by another organization, internal organizations within an enterprise, or product companies that provide commercial services used by an application.

Service Provider Design Considerations

When designing service-based applications, there are general guidelines that apply to all services. 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 Application (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 may be designing an Operational Data Store (ODS) that provides a service based interface. The design considerations in this section begin with general guidelines that apply to all services. Specific guidelines for different service types follow these.

General

Use the following guidelines, which discuss general areas of your design, to create an architecture that follows best practices and maximizes extensibility:

- **Design coarse-grained operations**. Avoid chatty calls to the service, 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, use standard elements to compose the complex types used by your service.
- **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 arrive out of order, implement a design that will store messages and then process them in the correct order.

SOA Services

Use the following service-oriented architecture guidelines to create a design that follows best practices for services:

- **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 sub-sets 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 that external consumers interact with. Inside the service implementation, you translate between internal business entities and external contracts.
- **Design services with explicit boundaries**. A service 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 assuming that malformed requests can be sent to a service.
- **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 a policy to determine interaction requirements.

Data Services

Use the following guidelines to design your architecture to follow best practices for data access and data services:

- **Avoid using services to expose individual tables in a database**. This will lead to chatty service calls with 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

Use the following guidelines to design your architecture to follow best practices for using workflow services:

- **Use interfaces supported by your workflow engine.** Attempting to create custom interfaces can restrict the type 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.

Service Consumer Design Guidelines

The consumer of a service will typically use service agents to access other services. A service agent represents a proxy that exposes operations and message contracts to the consumer and manages the communication with services, and translation of message contracts to and from the form used to serialize data across boundaries. From a consumer perspective, the service agent is just another component that can be used like any other component. However, when accessing services there are design guidelines that should be considered by the consumer of the service.

- **Design for connectivity related issues.** Service-based applications will generally rely on good quality broadband Internet connections to function well. However, in some cases, a service may not be available or the connection between the consumer and service may be unreliable. As a result, you should never assume that a service will always be available or reliable. Services that require large data transfers, such as backup services and file delivery services will generally run more slowly over an Internet connection compared to a local or in-house implementation.

- **Consider identity management in your design.** If a service depends on user identity, the provisioning and de-provisioning of user accounts must be extended to the service. Enterprise user account policies such as password complexity and account lockouts must be compatible with those of the service provider. Translation of in-house user identity into specific roles may be required, possibly through a federated service, to minimize the spread of individual user identities to a service.
- **Consider security requirements in your design.** Authentication, encryption, and the use of digital signatures may require the purchase of certificates from certified providers. Many organizations implement a Public Key Infrastructure (PKI) that require the installation of certificates on the consumer tier in order to interact with services provided by an external organization. Rules for actions that users can execute, such as limits on transaction size and other business rules must be maintained, even if these are not part of the service capabilities. This may make the service integration infrastructure more complex.
- **Design for differences in data formats and structure.** Requirements for data operations, such as Extract, Transform, and Load (ETL) and data integration, must be analyzed for compatibility with service capabilities. If required, you should plan for how you will migrate data to the service provider, and how you will migrate it away and to a different provider should the need arise.
- **Design for contract management.** Skills and expertise will be required to assess suppliers more comprehensively, and make choices regarding service acquisition and contracts. Service Level Agreements (SLAs) may require revision to ensure that they can still be met when depending on the services hosted by a remote provider.
- **Consider compliance and legal obligations in your design.** Legal compliance of the enterprise may be affected by the performance against compliance directives and legal obligations of the service provider. Compliance directives and legal obligations may differ if the service provider is located in another country or region. There may be costs associated with obtaining compliance reports from the service provider.

Services in the Cloud

As the use of services in application design has evolved the way that services are used has also evolved. Instead of just using services as an interface layer between components in a design, application and service developers are starting to take advantage of the Internet to distribute and access services. The Internet is commonly referred to as “the cloud”, which is where the term “Services in the Cloud” comes from. There are three main patterns associated with services in the cloud; software developed using services, services that provide application functionality, and services that provide platform functionality.

Software using Services

Software using services is an approach to applications are developed that combines hosted services with locally executed software. The remote services run over the Internet in what is usually termed "the cloud", hence the commonly used description "cloud computing". These services are consumed by software that is more directly suited to running locally on the user's machine, which may be a PC or any other Internet-enabled device. The combination of the

remote services and the software running locally provides a rich, seamlessly integrated user experience and a more comprehensive solution than traditional multi-tiered applications.

Application Services

Application services in their most simplistic form describe software deployed as a hosted service, and accessed over internet. A simple example could be a mail service hosted by an ISV, who manages the logic and the data, exposed to the consumers over the Internet.

Most of the components found in an application service are the same as would be found in any typical architecture. The services expose interfaces that smart clients and/or the Web presentation layer can invoke. They can initiate a synchronous workflow or a long-running transaction that will invoke other business services, which interact with the respective data stores to read and write business data. Security services are responsible for controlling access to end-user and back-end software services.

The most significant difference is the addition of metadata services, which are responsible for managing application configuration for individual tenants. Services and smart clients interact with the metadata services to retrieve information that describes the configuration and extensions specific to each tenant.

Platform Services

Services can be used to provide the entire infrastructure required to support end-to-end development and delivery of Web applications and web services. It provides a core hosting operating system, and optionally plug-in business services, that allow you to run your own applications or third-party applications obtained from vendors, on a remote cloud-based system.

Web applications may require pulling live data from external sources of the Internet. Service platforms allow communication with these different sources and make it easy to combine them into a single application, because every resource is Web-based. Such applications implemented on the platform need to connect to external sources automatically; this can be achieved by effective state management strategy when building platform services. Building effective platform services with information from multiple sources enable robust multi-tenancy.

patterns & practices Solution Assets

For more information on service interface layer design, message design, and versioning see the following sections of the *Web Service Software Factory: Modeling Edition* guidance:

- *Message Design in Service-Oriented Applications* at http://msdn.microsoft.com/en-us/library/cc304864.aspx.
- *Service Interface Layer* at http://msdn.microsoft.com/en-us/library/cc304737.aspx.
- *Useful Patterns for Services* at http://msdn.microsoft.com/en-us/library/cc304800.aspx.
- *Versioning Web Services* at http://msdn.microsoft.com/en-us/library/cc304741.aspx.

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 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 3 – Service Layer Guidelines

Objectives

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

Overview

When providing application functionality through services, it is important to separate the service functionality into a separate service layer. Within the service layer, you define the service interface, implement the service interface, and provide translator components that translate data formats between the business layer and external data contracts. One of the more important concepts to keep in mind is that a service should never expose internal entities that are used by the business layer. Figure 1. shows where a service layer fits in the overall design of your application.

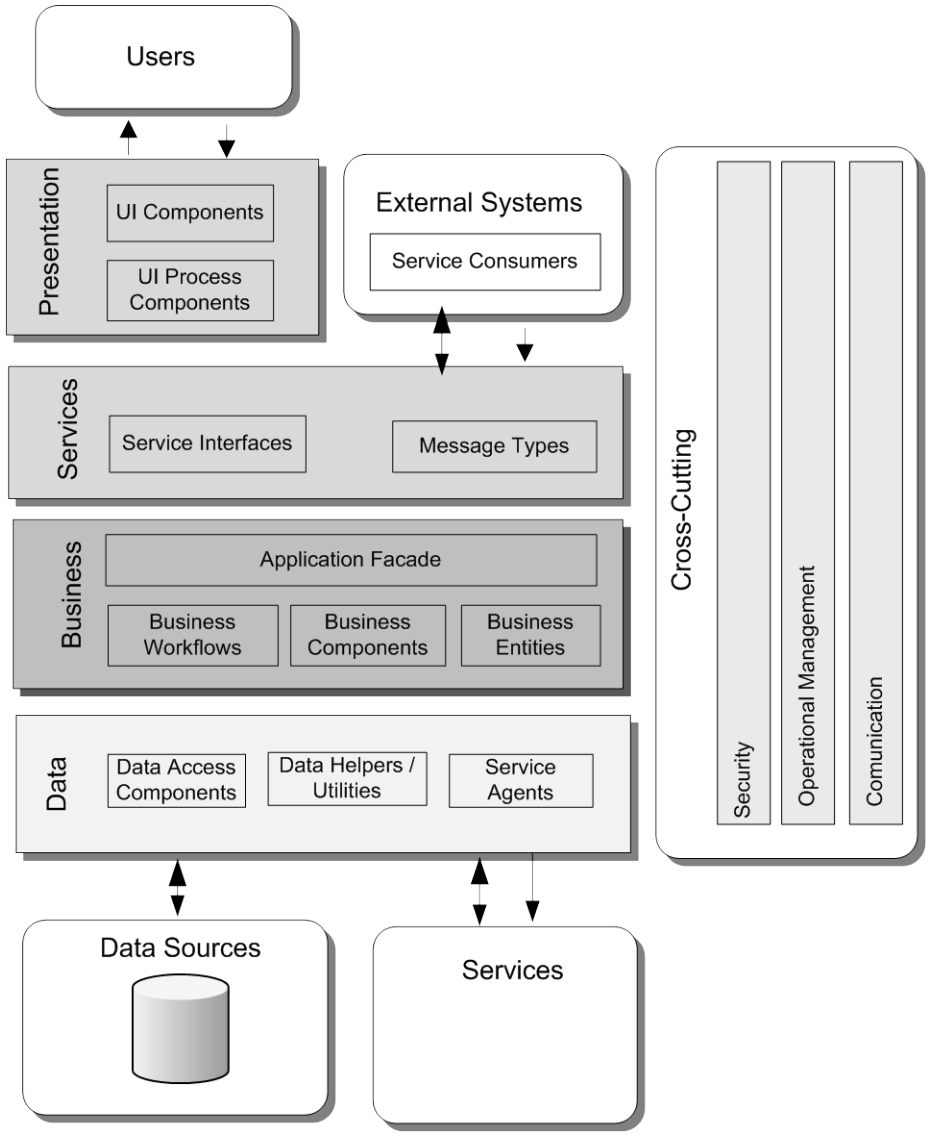

Figure 1 - An overall view of a typical application showing the service layer.

Service Layer Components

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

Approach

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

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

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

Design Considerations

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

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

- **Design to assume the possibility of invalid requests**. You should never assume that all messages received by the service are valid.
- **Separate functional business concerns from infrastructure operational concerns**. Cross cutting logic should never be combined with application logic. Doing so can lead to implementations that are difficult to extend and maintain.
- **Ensure that the service can detect and manage repeated messages (idempotency).** When designing the service, implement well-known patterns to ensure that duplicate messages are not processed.
- **Ensure that the service can manage messages arriving out of order (commutativity)**. If there is a possibility that messages arrive out of order, implement a design that will store messages and then process them in the correct order.
- **Versioning of Contracts.** A new version for service contracts mean new operations exposed by the service whereas for data contracts it means new schema type definitions being added.

Services Layer Frame

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

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

	• Not using transport layer protection for messages that cross multiple servers. • Not considering data integrity.
Message Routing	• Not choosing the appropriate router design. • Ability to access a specific item from a message. • Ensuring that messages are handled in the correct order.
Message Transformation	• Performing unnecessary transformations. • Implementing transformations at the wrong point. • Using a canonical model when not necessary.
REST	• There is limited schema support. • Current tools for REST are primitive. • Using hypertext to manage state.
SOAP	• Not choosing the appropriate security model. • Not planning for fault conditions. • Using complex types in the message schema.

Authentication

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

When designing an authentication strategy, consider following guidelines:

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

Authorization

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

When designing an authorization strategy, consider following guidelines:

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

Communication

When designing the communication strategy for your service, the protocol you choose should be based on the deployment scenario your service must support. If the service will be deployed within a closed network, you can use TCP for more efficient communications. If the service will be deployed in to a public facing network, you should choose the HTTP protocol.

When designing a communication strategy, consider following guidelines:

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

Data Consistency

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

When designing for data consistency, consider following guidelines:

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

Exception Management

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

When designing an exception management strategy, consider following guidelines:

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

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

Messaging Channels

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

When designing message channels, consider following guidelines:

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

Message Construction

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

When designing a message construction strategy, consider following guidelines:

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

Message Endpoint

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

When designing message endpoints, consider following guidelines:

- Determine relevant patterns for message endpoints such as Gateway, Mapper, Competing Consumers, and Message Dispatcher.
- Determine if you should accept all messages, or implement a filter to handle specific messages.
- Design for idempotency in your message interface. Idempotency is the situation where you could receive duplicate messages from the same consumer, but should only handle one. In

other words, an idempotent endpoint will guarantee that only one message will be handled, and all duplicate messages will be ignored.
- Design for commutativity in your message interface. Commutativity is related to the order that messages are received. In some cases, you may need to store inbound messages so that they can be processed in the correct order.
- Design for disconnected scenarios. For instance, you may need to support guaranteed delivery.

Message Protection

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

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

Message Routing

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

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

Message Transformation

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

When designing message transformation, consider following guidelines:

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

Representational State Transfer (REST)

Representational state transfer (REST) represents an architecture style for distributed systems. It is designed to reduce complexity by dividing a system into resources. The operations supported by a resource represent the functionality provided by a service that uses REST.

When designing REST resources, consider following guidelines:

- Identify and categorize resources that will be available to clients.
- Choose an approach for resource representation. A good practice would be to use meaningful names for REST starting points and unique identifiers, such as a GUID, for specific resource instances. For example, http://www.contoso.com/employee/8ce762d5-b421-6123-a041-5fbd07321bac4 represents an employee starting point while with a GUID that represents a specific employee appended to it.
- Decide if multiple views should be supported for different resources. For example, decide if the resource should support GET and POST operations, or only GET operations.

Service Interface

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

When designing a service interface, consider following guidelines:

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

- Design services interfaces in such a way that changes to the business logic do not affect the interface.
- Do not implement business rules in a service interface.
- Use standard formats for parameters to provide maximum compatibility with different types of client.
- Do not make assumptions in your interface design about the way that clients will use the service.
- Do not use object inheritance to implement versioning for the service interface.

SOAP

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

When designing SOAP messages, consider following guidelines:

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

Deployment Considerations

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

When deploying the service layer, consider following guidelines:

- Deploy the service layer to the same tier as the business layer to improve application performance unless performance and security issues inherent within the production environment prevent this.
- If the service is located on the same physical tier as the service consumer, consider using named pipes or shared memory protocols.
- If the service is accessed only by other applications within a local network, consider using TCP for communications.
- If the service is publicly accessible from the Internet, use HTTP for your transport protocol.

Pattern Map

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

	• Request Response
Data Consistency	• Atomic Transactions • Cross-service Transactions • Long running transactions
Messaging Channels	• Channel Adapter • Message Bus • Messaging Bridge • Point-to-point Channel • Publish-subscribe Channel
Message Construction	• Command Message • Document Message • Event Message • Request-Reply
Message Endpoint	• Competing Consumer • Durable Subscriber • Idempotent Receiver • Message Dispatcher • Messaging Gateway • Messaging Mapper • Polling Consumer • Selective Consumer • Service Activator • Transactional Client
Message Protection	• Data Confidentiality • Data Integrity • Data Origin Authentication • Exception Shielding • Federation • Replay Protection • Validation
Message Routing	• Aggregator • Content-Based Router • Dynamic Router • Message Broker (Hub-and-Spoke) • Message Filter • Process Manager
Message Transformation	• Canonical Data Mapper • Claim Check • Content Enricher • Content Filter • Envelope Wrapper • Normalizer

REST	• Behavior • Container • Entity • Store • Transaction
Service Interface	• Remote Façade
SOAP	• Data Contracts • Fault Contracts • Service Contracts

Pattern Descriptions

- **Aggregator -** A filter that collects and stores individual related messages, combines these messages, and publishes a single aggregated message to the output channel for further processing.
- **Atomic Transactions -** Transactions that are scoped to a single service operation.
- **Behavior -** (REST) Applies to resources that carry out operations. These resources generally contain no state of their own, and only support the POST operation.
- **Canonical Data Mapper -** Use a common data format to perform translations between two disparate data formats.
- **Channel Adapter -** A component that can access the application's API or data and publish messages on a channel based on this data, and can receive messages and invoke functionality inside the application.
- **Claim Check -** Retrieve data from a persistent store when required.
- **Command Message -** A message structure used to support commands.
- **Competing Consumer -** Set multiple consumers on a single message queue and have them compete for the right to process the messages, which allows the messaging client to process multiple messages concurrently.
- **Container -** Builds on the entity pattern by providing the means to dynamically add and/or update nested resources.
- **Content Enricher -** A component that enriches messages with missing information obtained from an external data source.
- **Content Filter -** Remove sensitive data from a message and reduce network traffic by removing unnecessary data from a message.
- **Content-Based Router -** Route each message to the correct consumer based on the contents of the message; such as existence of fields, specified field values, and so on.
- **Cross-service Transactions -** Transactions that can span multiple services.
- **Data Confidentiality -** Use message-based encryption to protect sensitive data in a message.
- **Data Contract -** A schema that defines data structures passed with a service request.
- **Data Integrity -** Ensure that messages have not been tampered with in transit.
- **Data Origin Authentication -** Validate the origin of a message as an advanced form of data integrity.

- **Document Message –** A structure used to reliably transfer documents or a data structure between application.
- **Duplex –** Two-way message communication where both the service and the client send messages to each other independently, irrespective of the use of the one-way or the request/reply pattern.
- **Durable Subscriber -** In a disconnected scenario, messages are saved and then made accessible to the client when connecting to the message channel to provide guaranteed delivery.
- **Dynamic Router -** A component that dynamically routes the message to a consumer after evaluating the conditions/rules that the consumer has specified.
- **Entity -** (REST) Resources that can be read with a GET operation, but can only be changed by PUT and DELETE operations.
- **Envelope Wrapper -** A wrapper for messages that contains header information used, for example, to protect, route, or authenticate a message.
- **Event Message -** A structure that provides reliable asynchronous event notification between applications.
- **Exception Shielding -** Prevent a service from exposing information about its internal implementation when an exception occurs.
- **Façade** – Implement a unified interface to a set of operations to provide a simplified reduce coupling between systems.
- **Fault Contracts -** A schema that defines errors or faults that can be returned from a service request.
- **Federation -** An integrated view of information distributed across multiple services and consumers.
- **Fire and Forget -** A one-way message communication mechanism used when no response is expected.
- **Idempotent Receiver -** Ensure that a service will only handle a message once.
- **Long-running Transaction -** Transactions that are part of a workflow process.
- **Message Broker (Hub-and-Spoke) -** A central component that communicates with multiple applications to receive messages from multiple sources, determine the correct destination, and route the message to the correct channel.
- **Message Bus -** Structure the connecting middleware between applications as a communication bus that enables them to work together using messaging.
- **Message Dispatcher -** A component that sends messages to multiple consumers.
- **Message Filter -** Eliminate undesired messages, based on a set of criteria, from being transmitted over a channel to a consumer.
- **Messaging Bridge -** A component that connects messaging systems and replicates messages between these systems.
- **Messaging Gateway -** Encapsulate message-based calls into a single interface in order to separate it from the rest of the application code.
- **Messaging Mapper -** Transform requests into business objects for incoming messages, and reverse the process to convert business objects into response messages.

- **Normalizer -** Convert or transform data into a common interchange format when organizations use different formats.
- **Point-to-point Channel -** Send a message on a Point-to-Point Channel to ensure that only one receiver will receive a particular message.
- **Polling Consumer -** A service consumer that checks the channel for messages at regular intervals.
- **Process Manager -** A component that enables routing of messages through multiple steps in a workflow.
- **Publish-subscribe Channel -** Create a mechanism to send messages only to the applications that are interested in receiving the messages without knowing the identity of the receivers.
- **Reliable Sessions -** End-to-end reliable transfer of messages between a source and a destination, regardless of the number or type of intermediaries that separate endpoints.
- **Remote Façade –** Create a high-level unified interface to a set of operations or processes in a remote subsystem to make that subsystem easier to use, by providing a course-grained interface over fine-grained operations to minimize calls across the network.
- **Replay Protection -** Enforce message idempotency by preventing an attacker from intercepting a message and executing it multiple times.
- **Request Response -** A two-way message communication mechanism where the client expects to receive a response for every message sent.
- **Request-Reply -** Use separate channels to send the request and reply.
- **Selective Consumer -** The service consumer uses filters to receive messages that match specific criteria.
- **Service Activator -** A service that receives asynchronous requests to invoke operations in business components.
- **Service Contract** – A schema that defines operations that the service can perform.
- **Service Interface** – A programmatic interface that other systems can use to interact with the service.
- **Store -** (REST) Allows entries to be created and updated with PUT.
- **Transaction -** (REST) Resources that support transactional operations.
- **Transactional Client -** A client that can implement transactions when interacting with a service.
- **Validation -** Check the content and values in messages to protect a service from malformed or malicious content.

Technology Considerations

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

- Consider using ASP.NET Web services (ASMX) for simplicity, but only when a suitable Web server will be available.
- Consider using Windows Communication Foundation (WCF) services for advanced features and support for multiple transport protocols.
- If you are using ASP.NET Web Services, and you require message-based security and binary data transfer, consider using Web Service Extensions (WSE).

- If you are using WCF and you want interoperability with non-WCF or non-Windows clients, consider using HTTP transport based on SOAP specifications.
- If you are using WCF and you want to support clients within an intranet, consider using the TCP protocol and binary message encoding with transport security and Windows authentication.
- If you are using WCF and you want to support WCF clients on the same machine, consider using the named pipes protocol and binary message encoding.
- If you are using WCF, consider defining service contracts that use an explicit message wrapper instead of an implicit one. This allows you to define message contracts as inputs and outputs for your operations, which then allows you to extend the data contracts included in the message contract without affecting the service contract.

Additional Resources

For more information, see the following resources:

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

Chapter 4 - Communication Guidelines

Objectives

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

Overview

One of the key factors that affect the design of an application, particularly a distributed application, is the way that you design the communication infrastructure for each part of the application. Components must communicate with each other, for example to send user input to the business layer, and then to update the data store through the data layer. When components are located on the same physical tier, you can often rely on direct communication between these components. However, if you deploy components and layers on physically separate servers and client machines - as is likely in most scenarios - you must consider how the components in these layers will communicate with each other efficiently and reliably.

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

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

Design Guidelines

When designing a communication strategy for your application, consider the performance impact of communicating between layers, as well as between tiers. Because each communication across a logical or a physical boundary increases processing overhead, design for efficient communication by reducing round trips and minimizing the amount of data sent over the network.

- **Consider communication strategies when crossing boundaries.** Understand each of your boundaries, and how they affect communication performance. For example, the application domain (AppDomain), computer process, machine, and unmanaged code all represent

boundaries that that can be crossed when communicating with components of the application or external services and applications.

- **Consider using unmanaged code for communication across AppDomain boundaries**. Use unmanaged code to communicate across AppDomain boundaries. This approach requires assemblies to run in full trust in order to interact with unmanaged code.
- **Consider using message-based communication when crossing process boundaries.** Use Windows Communication Foundation (WCF) with either the TCP or named pipes protocols to package data into a single call that can be serialized across process boundaries.
- **Consider message-based communication when crossing physical boundaries.** Consider using Windows Communication Foundation (WCF) or Microsoft Message Queuing (MSMQ) to communicate with remote machines across physical boundaries. Message-based communication supports coarse-grained operations that reduce round trips when communicating across a network.
- **Reduce round trips when accessing remote layers**. When communicating with remote layers, reduce communication requirements by using coarse-grained message-based communication methods, and use asynchronous communication if possible to avoid blocking or freezing the user interface.
- **Consider the serialization capabilities of the data formats passed across boundaries**. If you require interoperability with other systems, consider XML serialization. Keep in mind that XML serialization imposes increased overhead. If performance is critical, consider binary serialization because it is faster and the resulting serialized data is smaller than the XML equivalent.
- **Consider hotspots while designing your communication policy.** Hotspots include asynchronous and synchronous communication, data format, communication protocol, security, performance, and interoperability.

Message-Based Communication

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

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

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

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

Asynchronous and Synchronous Communication

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

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

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

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

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

- Consider using BizTalk Server to interoperate with other systems and platforms at enterprise level, or for Electronic Data Interchange (EDI).

Coupling and Cohesion

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

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

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

State Management

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

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

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

Message Format

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

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

- Ensure that type information is not lost during the communication process. Binary serialization preserves type fidelity, which is useful when passing objects between client

and server. Default XML serialization serializes only public properties and fields and does not preserve type fidelity.
- Ensure that your application code can detect and manage messages that arrive more than once (idempotency).
- Ensure that your application code can detect and manage multiple messages that arrive out of order (commutativity).

Passing Data Through Tiers - Data Formats

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

- Consider the advantage of using custom objects; these can impose a lower overhead than DataSets and support both binary and XML serialization.
- If your application works mainly with sets of data, and needs functionality such as sorting, searching and data binding, consider using DataSets. Consider that DataSets introduce serialization overhead.
- If your application works mainly with instance data, consider using scalar values for better performance.

Data Format Considerations

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

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

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

	• You need to handle sets and complex relationships. • You need to track changes to data within the DataSet.
Custom Objects	• You need support for complex data structures. • You are communicating with components that know about the object type. • You want to support binary serialization for performance.

Interoperability Considerations

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

Consider the following guidelines for maximizing interoperability:

- To enable communication with wide variety of platforms and devices, consider using standard protocols such as SOAP or REST. With both protocols, the structure of message data is defined using XML schemas.
- Keep in mind that protocol decisions may affect the availability of clients you are targeting. For example, target systems may be protected by firewalls that block some protocols.
- Keep in mind that data format decision may affect interoperability. For example, target systems may not understand platform-specific types, or may have different ways of handling and serializing types.
- Keep in mind that security encryption and decryption decisions may affect interoperability. For example, some message encryption/decryption techniques may not be available on all systems.

Performance Considerations

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

Consider the following guidelines for performance:

- Avoid fine-grained "chatty" interfaces for cross-process and cross-machine communication. These require the client to make multiple method calls to perform a single logical unit of work. Consider using the Façade pattern to provide a coarse-grained wrapper for existing chatty interfaces.
- Use Data Transfer Objects to pass data as a single unit instead of passing individual data types one at a time.
- Reduce the volume of data passed to remote methods where possible. This reduces serialization overhead and network latency.
- If serialization performance is critical for your application, consider using custom classes with binary serialization.

- If XML is required for interoperability, use attribute based structures for large amounts of data instead of element based structures.

Security Considerations

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

Transport Security.

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

Consider the following when deciding to use transport security:

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

Message Security

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

Consider the following guidelines for message security:

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

WCF Technology Options

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

- You can use WCF to communicate with Web services to achieve interoperability with other platforms that also support SOAP, such as the J2EE-based application servers.
- You can use WCF to communicate with Web services using messages not based on SOAP for applications with formats such as RSS.
- You can use WCF to communicate using SOAP messages and binary encoding for data structures when both the server and the client use WCF.
- You can use WS-MetadataExchange in SOAP requests to obtain descriptive information about a service, such as its WSDL definition and policies.
- You can use WS-Security to implement authentication, data integrity, data privacy, and other security features.
- You can use WS-Reliable Messaging to implement reliable end-to-end communication, even when one or more Web services intermediaries must be traversed.
- You can use WS-Coordination to coordinate two-phase commit transactions in the context of Web services conversations.
- You can use WCF to build REST Singleton & Collection Services, ATOM Feed and Publishing Protocol Services, and HTTP Plain XML Services.

WCF supports several different protocols for communication:

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

ASMX Technology Options

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

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

- ASPX services support the endpoint technology implemented in IIS.
- ASPX services provide cross-platform interoperability and cross-company computing.

REST vs. SOAP

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

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

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

Checklist - Service Layer

Design Considerations

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

Authentication

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

Authorization

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

Communication

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

Data Consistency

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

Exception Management

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

Message Channels

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

Message Construction

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

Message Endpoint

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

Message Protection

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

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

Message Routing

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

Message Transformation

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

Representational State Transfer (REST)

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

Service Interface

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

SOAP

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

Deployment Considerations

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

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