Chapter 7: Business Layer Guidelines

01/26/2010 • 16 minutes to read

In this article

For more details of the topics covered in this guide, see Contents of the Guide.

Contents

Overview

This chapter describes the key guidelines for designing the business layer of an application. It will help you to understand how the business layer fits into the typical layered application architecture, the components it usually contains, and the key issues you face when designing the business layer. You will see guidelines for design, the recommended design steps, relevant design patterns, and technology options. Figure 1 shows how the business layer fits into a typical application architecture.

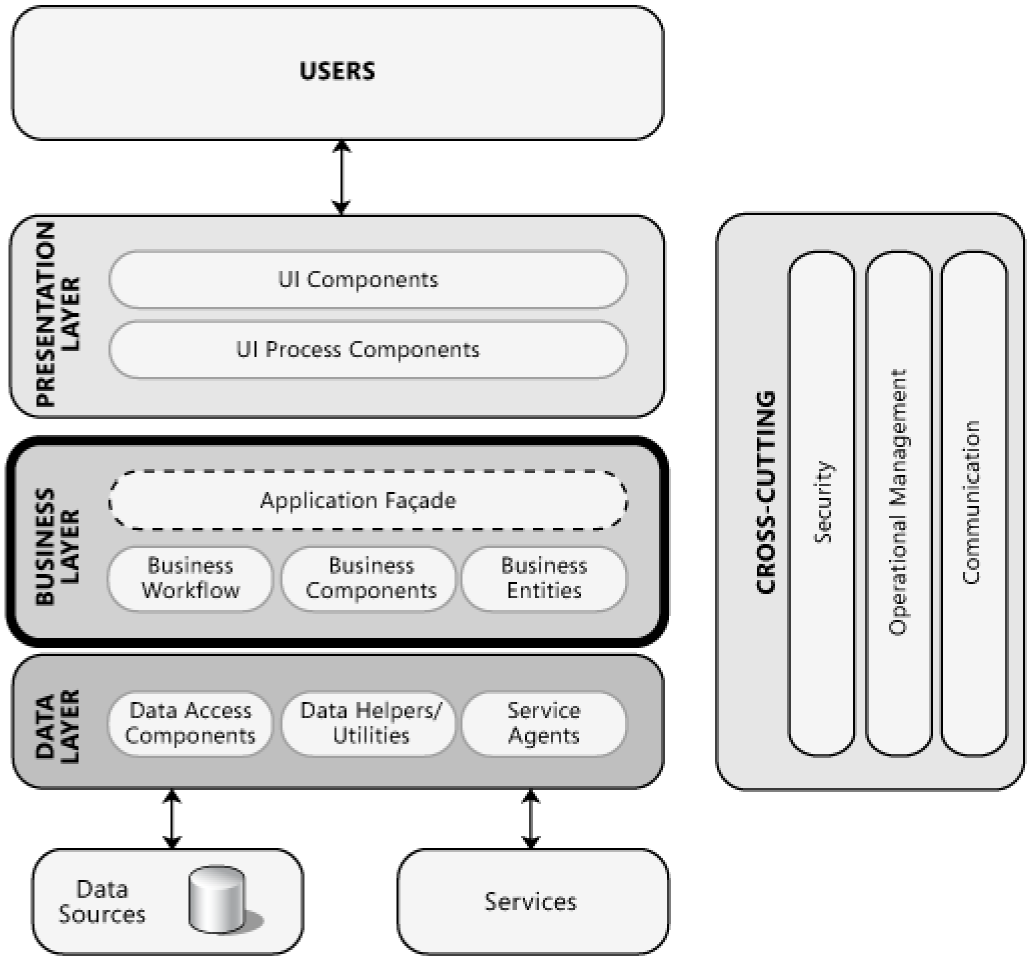

Figure 1

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

The business layer will usually include the following:

- **Application façade.** This optional component typically provides a simplified interface to the business logic components, often by combining multiple business operations into a single operation that makes it easier to use the business logic. It reduces dependencies because external callers do not need to know details of the business components and the relationships between them.
- **Business Logic components**. Business logic is defined as any application logic that is concerned with the retrieval, processing, transformation, and management of application data; application of business rules and policies; and ensuring data consistency and validity. To maximize reuse opportunities, business logic components should not contain any behavior or application logic that is specific to a use case or user story. Business logic components can be further subdivided into the following two categories:
    - **Business Workflow components**. After the UI components collect the required data from the user and pass it to the business layer, the application can use this data to perform a business process. Many business processes involve multiple steps that must be performed in the correct order, and may interact with each other through an orchestration. Business workflow components define and coordinate long running, multistep business processes, and can be implemented using business process management tools. They work with business process components that instantiate and perform operations on workflow components. For more information on business workflow components, see Chapter 14 "Designing Workflow Components."
    - **Business Entity components**. Business entities, or—more generally—business objects, encapsulate the business logic and data necessary to represent real world elements, such as Customers or Orders, within your application. They store data values and expose them through properties; contain and manage business data used by the application; and provide stateful programmatic access to the business data and related functionality. Business entities also validate the data contained within the entity and encapsulate business logic to ensure consistency and to implement business rules and behavior. For more information about business entity components, see Chapter 13 "Designing Business Entities."

For more information about the components commonly used in the business layer, see Chapter 10 "Component Guidelines."

For more information about designing components for the business layer, see Chapter 12 "Designing Business Components."

General Design Considerations

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

- **Decide if you need a separate business layer**. It is always a good idea to use a separate business layer where possible to improve the maintainability of your application. The exception may be applications that have few or no business rules (other than data validation).
- **Identify the responsibilities and consumers of your business layer**. This will help you to decide what tasks the business layer must accomplish, and how you will expose your business layer. Use a business layer for processing complex business rules, transforming data, applying policies, and for validation. If your business layer will be used by your presentation layer and by an external application, you may choose to expose your business layer through a service.
- **Do not mix different types of components in your business layer**. Use a business layer to avoid mixing presentation and data access code with business logic code, to decouple business logic from presentation and data access logic, and to simplify testing of business functionality. Also, use a business layer to centralize common business logic functions and promote reuse.
- **Reduce round trips when accessing a remote business layer**. If the business layer is on a separate physical tier from layers and clients with which it must interact, consider implementing a message-based remote application façade or service layer that combines fine-grained operations into a smaller number of coarse-grained operations. Consider using coarse-grained packages for data transported over the network, such as Data Transfer Objects (DTOs).
- **Avoid tight coupling between layers**. Use the principles of abstraction to minimize coupling when creating an interface for the business layer. Techniques for abstraction include using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation layer and the business layer. For more details, see Chapter 5 "Layered Application Guidelines."

Specific Design Issues

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 sections provide guidelines for the common areas where mistakes are most often made:

- Authentication
- Authorization
- Caching
- Coupling and Cohesion
- Exception Management
- Logging, Auditing, and Instrumentation
- Validation

Authentication

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

- Avoid authentication in the business layer if it will be used only by a presentation layer or by a service layer on the same tier within a trusted boundary. Flow the caller's identity to the business layer only if you must authenticate or authorize based on the original caller's ID.
- If your business layer will be used in multiple applications, using separate user stores, consider implementing a single sign-on mechanism. Avoid designing custom authentication mechanisms; instead, make use of the built-in platform mechanisms whenever possible.
- If the presentation and business layers are deployed on the same machine and you must access resources based on the original caller's access control list (ACL) permissions, consider using impersonation. If the presentation and business layers are deployed to separate machines and you must access resources based on the original caller's ACL permissions,

consider using delegation. However, use delegation only when necessary due to the increased use of resources, and additionally, because many environments do not support it. If your security requirements allow, consider authenticating the user at the boundary and using the trusted subsystem approach for calls to lower layers. Alternatively, consider using a claims-based security approach (especially for service-based applications) that takes advantage of federated identity mechanisms and allows target system to authenticate the user's claims.

Authorization

Designing an effective authorization strategy for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges. Consider the following guidelines when designing an authorization strategy:

- Protect resources by applying authorization to callers based on their identity, account groups, roles, or other contextual information. For roles, consider minimizing the granularity of roles as far as possible to reduce the number of permission combinations required.
- Consider using role-based authorization for business decisions; resource-based authorization for system auditing; and claims-based authorization when you need to support federated authorization based on a mixture of information such as identity, role, permissions, rights, and other factors.
- Avoid using impersonation and delegation where possible because it can significantly affect performance and scaling opportunities. It is generally more expensive to impersonate a client on a call than to make the call directly.
- Do not mix authorization code and business processing code in the same components.
- As authorization is typically pervasive throughout the application, ensure that your authorization infrastructure does not impose any significant performance overhead.

Caching

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

- Consider caching static data that will be reused regularly within the business layer, but avoid caching volatile data. Consider caching data that cannot be retrieved from the database quickly and efficiently, but avoid caching very large volumes of data that can slow down processing. Cache only the minimum required.
- Consider caching data in a ready to use format within your business layer.
- Avoid caching sensitive data if possible, or design a mechanism to protect sensitive data in the cache.
- Consider how Web farm deployment will affect the design of your business layer caching solution. If any server in the farm can handle requests from the same client, your caching solution must support the synchronization of cached data.

For more information on caching techniques, see Chapter 17 "Crosscutting Concerns."

Coupling and Cohesion

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

- Avoid circular dependencies. The business layer should know only about the layer below (the data access layer), and not the layer above (the presentation layer or external applications that access the business layer directly).
- Use abstraction to implement a loosely coupled interface. This can be achieved with interface components, common interface definitions, or shared abstraction where concrete components depend on abstractions and not on other concrete components (the principle of Dependency Inversion). For more information, see the steps for designing a layered structure in Chapter 5 "Layered Application Guidelines."
- Design for tight coupling within the business layer unless dynamic behavior requires loose coupling.
- Design for high cohesion. Components should contain only functionality specifically related to that component. Always avoid mixing data access logic with business logic in your business components.
- Consider using message-based interfaces to expose business components to reduce coupling and allow them to be located on separate physical tiers if required.

Exception Management

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

- Only catch internal exceptions that you can handle, or if you need to add information. For example, catch data conversion exceptions that can occur when trying to convert null values. Do not use exceptions to control business logic or application flow.
- Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer. Consider including a context identifier so that related exceptions can be associated across layers when performing root cause analysis of errors and faults.
- Ensure that you catch exceptions that will not be caught elsewhere (such as in a global error handler), and clean up resources and state after an exception occurs.
- Design an appropriate logging and notification strategy for critical errors and exceptions that logs sufficient detail from exceptions and does not reveal sensitive information.

For more information on exception management techniques, see Chapter 17 "Crosscutting Concerns."

Logging, Auditing, and Instrumentation

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

- Centralize the logging, auditing, and instrumentation for your business layer. Consider using a library such as patterns & practices Enterprise Library, or a third party solutions such as the Apache Logging Services "log4Net" or JarosÅ,aw Kowalski's "NLog," to implement exception handling and logging features.
- Include instrumentation for system critical and business critical events in your business components.
- Do not store business sensitive information in the log files.
- Ensure that a logging failure does not affect normal business layer functionality.
- Consider auditing and logging all access to functions within business layer.

Validation

Designing an effective validation solution for your business layer is important for the usability and reliability of your application. Failure to do so can leave your application open to data inconsistencies and business rule violations, and a poor user experience. In addition, it may leave your application vulnerable to security issues such as cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attacks. There is no comprehensive definition of what constitutes a valid input or malicious input. In addition, how your application uses input influences the risk of the exploit. Consider the following guidelines when designing a validation strategy:

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

Deployment Considerations

When deploying a business layer, you must consider performance and security issues within the production environment. Consider the following guidelines when deploying a business layer:

- Consider deploying the business layer on the same physical tier as the presentation layer in order to maximize application performance, unless you must use a separate tier due to scalability or security concerns.
- If you must support a remote business layer, consider using the TCP protocol to improve application performance.
- Consider using Internet Protocol Security (IPSec) to protect data passed between physical tiers.
- Consider using Secure Sockets Layer (SSL) encryption to protect calls from business layer components to remote Web services.

Design Steps for the Business Layer

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

1. **Create a high level design for your business layer**. Identify the consumers of your business layer, such as the presentation layer, a service layer, or other applications. This will help you to determine how to expose your business layer. Next, determine the security requirements for your business layer, and the validation requirements and validation strategy. Use the guidelines in the "Specific Design Issues" section earlier in this chapter to ensure that you consider all of the relevant factors when creating the high level design.
2. **Design your business components**. There are several types of business components you can use when designing and implementing an application. Examples of these components include business process components, utility components, and helper components. Different aspects of your application design, transactional requirements, and processing rules affect the design you choose for your business components. For more information, see Chapter 12 "Designing Business Components."
3. **Design your business entity components**. Business entities are used to contain and manage business data used by an application. Business entities should provide validation of the data contained within the entity. In addition, business

entities provide properties and operations used to access and initialize data contained within the entity. For more information, see Chapter 13 "Designing Business Entities."

4. **Design your workflow components**. There are many scenarios where tasks must be completed in an ordered way based on the completion of specific steps, or coordinated through human interaction. These requirements can be mapped to key workflow scenarios. You must understand how requirements and rules affect your options for implementing workflow components. For more information, see Chapter 14 "Designing Workflow Components."

For more information about designing and using components in your applications, see Chapter 10 "Component Guidelines."

Relevant Design Patterns

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

Category	Relevant patterns
Business Components	**Application Façade**. Centralize and aggregate behavior to provide a uniform service layer. **Chain of Responsibility**. Avoid coupling the sender of a request to its receiver by allowing more than one object to handle the request. **Command**. Encapsulate request processing in a separate command object with a common execution interface.
Business Entities	**Domain Model**. A set of business objects that represents the entities in a domain and the relationships between them. **Entity Translator**. An object that transforms message data types to business types for requests, and reverses the transformation for responses. **Table Module**. A single component that handles the business logic for all rows in a database table or view.

Category	Relevant patterns
Workflows	**Data-Driven Workflow**. A workflow that contains tasks whose sequence is determined by the values of data in the workflow or the system. **Human Workflow**. A workflow that involves tasks performed manually by humans. **Sequential Workflow**. A workflow that contains tasks that follow a sequence, where one task is initiated after completion of the preceding task. **State-Driven Workflow**. A workflow that contains tasks whose sequence is determined by the state of the system.

For more information on the Façade pattern, see Chapter 4, "Structural Patterns" in Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, 1995.

For more information on the Chain of Responsibility pattern, see "*Patterns in Practice*" at http://msdn.microsoft.com/en-us/magazine/cc546578.aspx.

For more information on the Command pattern, see 5, "Behavioral Patterns" in Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, 1995.

For more information on the Entity Translator pattern, see "*Useful Patterns for Services*" at http://msdn.microsoft.com/en-us/library/cc304800.aspx.

For more information on the Data-Driven Workflow, Human Workflow, Sequential Workflow, and State-Driven Workflow, see "*Windows Workflow Foundation Overview*" at http://msdn.microsoft.com/en-us/library/ms734631.aspx and "*Workflow Patterns*" at http://www.workflowpatterns.com/.

patterns & practices Offerings

For more information on relevant offerings available from the Microsoft patterns & practices group, see the following resources:

- "*Enterprise Library*" at http://msdn.microsoft.com/en-us/library/cc467894.aspx.
- "*Unity*" (dependency injection mechanism) at http://msdn.microsoft.com/en-us/library/dd203101.aspx.

Additional Resources

For more information on integrating business layers, see "*Integration Patterns*" at http://msdn.microsoft.com/en-us/library/ms978729.aspx.

For more information on Apache Logging Services "log4Net," see http://logging.apache.org/log4net/index.html.

For more information on JarosÅ‚aw Kowalski's "NLog," see http://nlog-project.org/home.