Web Application Architecture

The core of a Web application is its server-side logic.

The Web application layer itself can comprise many distinct layers.

The typical example is a three-layered architecture consisting of presentation, business, and data layers.

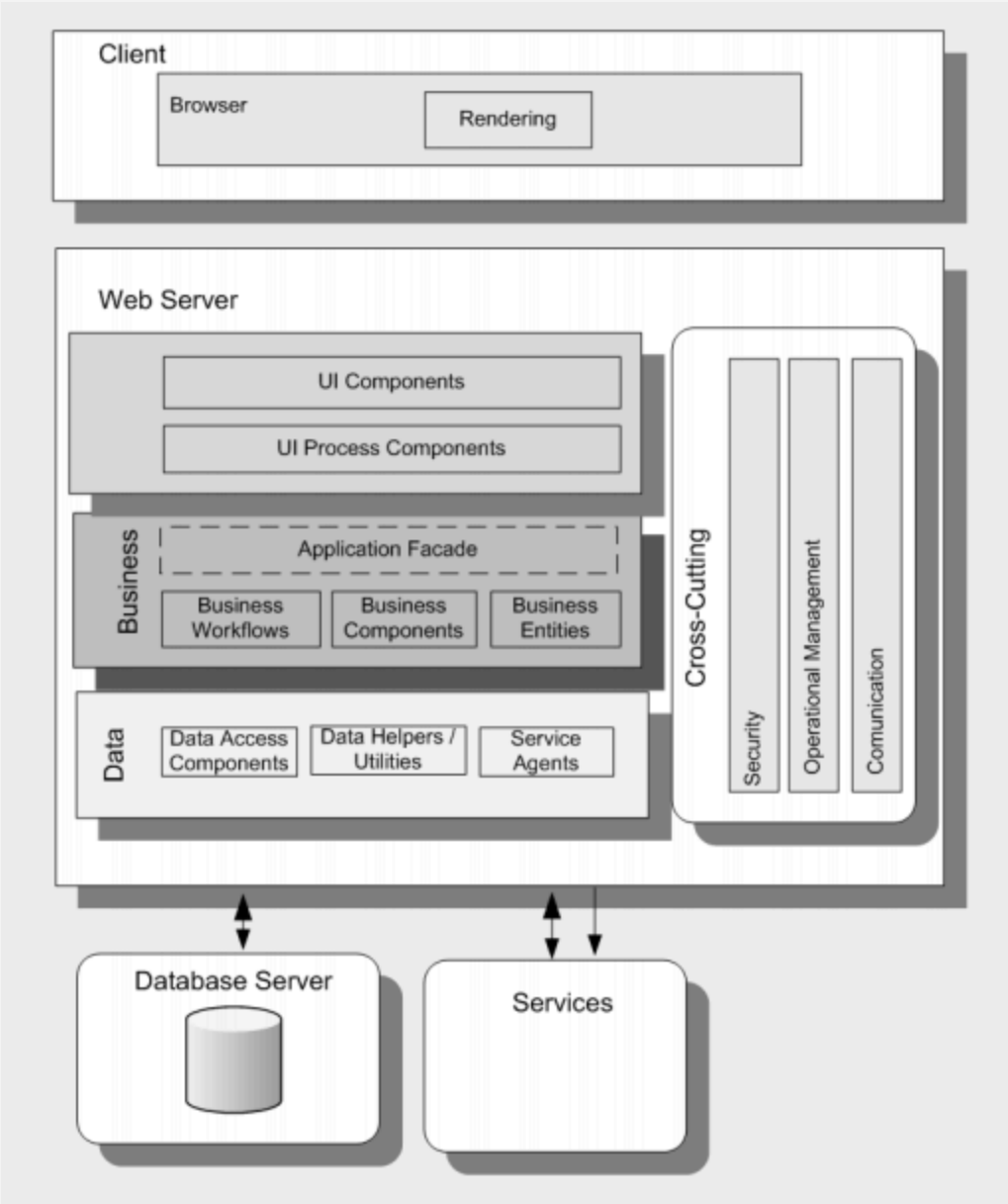

Figure 1. A common Web application architecture

Figure 1 illustrates a common Web application architecture with common components grouped by different areas of concern.

Design Considerations

When designing a Web application, the goals of a software architect are to minimize the complexity by separating tasks into different areas of concern while designing a secure and high performance application. When designing Web application, consider following guidelines:

- **Partition your application logically.**

Use layering to partition your application logically into presentation, business, and data access layers. This helps you to create maintainable code and allows you to monitor and optimize the performance of each layer separately. A clear logical separation also offers more choices for scaling your application.

- **Use abstraction to implement loose coupling between layers**.

This can be accomplished by defining interface components, such as a façade with well-known inputs and outputs that translates requests into a format understood by components within the layer. In addition, you can also use Interface types or abstract base classes to define a shared abstraction that must be implemented by interface components.

• **Understand how components will communicate with each other**.
This requires an understanding of the deployment scenarios your application must support. You must determine if communication across physical boundaries or process boundaries should be supported, or if all components will run within the same process.

• **Reduce round trips.**
When designing a Web application, consider using techniques such as caching and output buffering to reduce round trips between the browser and the Web server, and between the Web server and downstream servers.

• **Consider using caching.**
A well-designed caching strategy is probably the single most important performance-related design consideration. ASP.NET caching features include output caching, partial

page caching, and the cache API. Design your application to take advantage of these features.

• **Consider using logging and instrumentation.**
You should audit and log activities across the layers and tiers of your application. These logs can be used to detect suspicious activity, which frequently provides early indications of an attack on the system.

• **Avoid blocking during long-running tasks.**
If you have long-running or blocking operations, consider using an asynchronous approach to allow the Web server to process other incoming requests.

• **Consider authenticating users across trust boundaries.**
You should design your application to authenticate users whenever they cross a trust boundary; for example, when accessing a remote business layer from your presentation layer.

• **Do not pass sensitive data in plain text across the network.**
Whenever you need to pass sensitive data such as a password or authentication cookie across the network, consider encrypting and signing the data or using SSL.

• **Design to run your Web application using a least-privilege account.**
If an attacker manages to take control of a process, the process identity should have restricted access to the file system and other system resources in order to limit the possible damage.

Web Application Frame

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

Category Key Issues

Authentication

- Lack of authentication across trust boundaries.
- Storing passwords in a database as plain text.
- Designing custom authentication mechanism instead of using built-in capabilities.

Authorization

- Lack of authorization across trust boundaries.
- Incorrect role granularity.
- Using impersonation and delegation when not required.

Caching

- Caching volatile data.
- Not considering caching page output.
- Caching sensitive data.
- Failing to cache data in a ready-to-use format.

Exception Management

- Revealing sensitive information to the end user.
- Not logging sufficient details about the exception.
- Using exceptions for application logic.

Logging and Instrumentation

- Failing to implement adequate instrumentation in all layers.
- Failing to log system-critical and business-critical events.
- Not supporting runtime configuration of logging and instrumentation.
- Logging sensitive information.

Navigation

- Mixing navigation logic with user interface components.
- Hard-coding relationships between views.
- Not verifying if the user is authorized to navigate to a view.

Page Layout (UI)

- Using table-based layout for complex layouts.
- Designing complex and overloaded pages.

Page Rendering

- Using excessive postbacks that impact user experience.
- Using excessive page sizes that reduce performance.

Presentation Entity

- Creating custom entity objects when not required.

- Adding business logic to presentation entities.

Request Processing

- Mixing processing and rendering logic.
- Choosing an inappropriate pattern.

Service Interface Layer

- Breaking the service interface.
- Implementing business rules in a service interface.
- Failing to consider interoperability requirements.

Session Management

- Using an incorrect state store.
- Not considering serialization requirements.
- Not persisting when required.
- Enabling view state for large data items such as Datasets

Validation

- Relying on client side validation.
- Lack of validation across trust boundaries.
- Not reusing the validation logic.

Authentication

Designing an effective authentication strategy is important for the security and reliability of your application.

Improper or weak authorization can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attack.

When designing an authentication strategy, consider following guidelines:

• Identify trust boundaries within Web application layers.
This will help you to determine where to authenticate.

• Use a platform-supported authentication mechanism such as Windows Authentication when possible.
• If you are using Forms authentication, use the platform features when possible.
• Enforce strong account management practices such as account lockouts and expirations.
• Enforce strong password policies.

This includes specifying password length and complexity, and password expiration policies.

Authorization

Authorization determines the tasks that an authenticated identity can perform and identifies the resources that can be accessed. Designing an effective authorization strategy is important for the security and reliability of your application. Improper or weak authorization leads to information disclosure, data tampering, and elevation of privileges. Defense in depth is the key security principle to apply to your application's authorization strategy.

When designing an authorization strategy, consider following guidelines:

• Identify trust boundaries within the Web application layers and authorize users across trust boundaries.
• Use URL authorization for page and directory access control.
• Consider the granularity of your authorization settings.

Too fine granularity increases management overheads and too coarse granularity reduces flexibility.

• Access downstream resources using a trusted identity based on the trusted sub-system model.
• Use impersonation and delegation to take advantage of the user-specific auditing and granular access controls of the platform, but consider the effect on performance and

scalability.

Caching

Caching improves the performance and responsiveness of your application. However, incorrect caching choices and poor caching design can degrade performance and responsiveness. You should use caching to optimize reference data lookups, avoid network round trips, and avoid unnecessary and duplicate processing. To implement caching, you must decide when to load the cache data. Try to load the cache asynchronously or by using a batch process to avoid client delays. When designing caching, consider following guidelines:

• Avoid caching volatile data.

• Use output caching to cache pages that are relatively static.

• Consider using partial page caching through user controls for static data in your pages.

• Pool shared resources that are expensive, such as network connections, instead of caching them.

• Cache data in a ready-to-use format.

Exception Management

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

When designing an exception management strategy, consider following guidelines:

• Do not use exceptions to control logic flow, and design your code to avoid exceptions where possible.
• Do not catch exceptions unless you can handle them or you need to add information to the exception.
• Design a global error handler to catch unhandled exceptions.

• Display user-friendly messages to end users whenever an error or exception occurs.

• Do not reveal sensitive information, such as passwords, through exception details.

Logging and Instrumentation

Designing an effective logging and instrumentation strategy is important for the security and reliability of your application. You should audit and log activity across the tiers of your application. These logs can be used to detect suspicious activity, which frequently provides early indications of an attack on the system, and help to address the repudiation threat where users deny their actions. Log files may be required in legal proceedings to prove the wrongdoing of individuals. Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access and by the same routines that access the resource.

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

• Consider auditing for user management events.

• Consider auditing for unusual activities.

• Consider auditing for business critical operations.

• Create secure log file management policies, such as restricting the access to log files, allowing only write access to users, etc.

• Do not store sensitive information in the log or audit files.

Navigation

Design your navigation strategy in a way that separates it from the processing logic. It should allow users to navigate easily through your screens or pages. Designing a consistent navigation structure for your application will help to minimize user confusion as well as reducing the apparent complexity of the application.

When designing your navigation strategy, consider the following guidelines:

- Use well-known design patterns, such as Model-View-Presenter (MVP), to decouple UI processing from output rendering.
- Consider encapsulating navigation in a Master Page so that it is consistent across pages.
- Design a site-map to help users find pages on the site, and to allow search engines to crawl the site if desired.
- Consider using wizards to implement navigation between forms in a predictable way.
- Consider using visual elements such as embedded links, navigation menus, and breadcrumb text in the UI to help users understand where they are, what is available on the site, and how to navigate the site quickly.

Page Layout (UI)

Design your application so that the page layout can be separated from the specific UI components and UI processing. When choosing a layout strategy, consider whether designers or developers will be building the layout. If designers will be building the layout, choose a layout approach that does not require coding or the use of development-focused tools.

When designing your layout strategy, consider the following guidelines:

• Use Cascading Style Sheets (CSS) for layout whenever possible.
• Use table-based layout when you need to support a grid layout, but remember that table based layout can be slow to render, does not have full cross browser support, and there may be issues with complex layout.

• Use a common layout for pages where possible to maximize accessibility and ease of use.
• Use Master Pages in ASP.NET applications to provide a common look and feel for all of the pages.
• Avoid designing and developing large pages that accomplish multiple tasks, particularly where usually only a few tasks are executed with each request.

Page Rendering

When designing for page rendering, you must ensure that you render the pages efficiently and maximize interface usability.

When designing a page rendering strategy, consider following guidelines:

- Consider data binding options. For example, you can bind custom objects or datasets to controls. However, be aware that binding only applies to rendered data in ASP.NET.
- Consider using AJAX for an improved user experience and better responsiveness.
- Consider using data paging techniques for large amounts of data to minimize scalability issues.
- Consider designing to support localization in user interface components.
- Abstract the user process components from data rendering and acquisition functions.

Presentation Entity

Presentation entities store the data that you will use to manage the views in your presentation layer. Presentation entities are not always necessary. Consider using presentation entities only if the data sets are sufficiently large or complex that they must be stored separately from the UI controls. Design or choose appropriate presentation entities that you can easily bind to user interface controls.

When designing presentation entities, consider the following guidelines:

• Determine if you need presentation entities, typically you might need presentations entities if the data or data format to be displayed is specific to the presentation layer.

• Consider the serialization requirements for your presentation entities, if they are to be passed across the network or stored on the disk.

• Consider implementing input data validation in your presentation entities.

• Consider using presentation entities to store state related to the user interface. If you want to use this state to help your application recover from a crash, make sure after recovery that the user interface is in a consistent state.

Request Processing

When designing a request processing strategy, you should ensure separation of concerns by implementing the request processing logic separately from the user interface. When designing a request processing strategy, consider the following guidelines:

• Consider centralizing the common pre-processing and post-processing steps of web page requests to promote logic reuse across pages. For example, consider creating a base class derived from the Page class to contain your common pre- and post-processing logic.

• Consider dividing UI processing into three distinct roles, model, view, and controller/presenter, by using the MVC or MVP pattern.

• If you are designing views for handling large amounts of data, consider giving access to the model from the view using the Supervising Controller pattern, which is a form of the MVP pattern.

• If your application does not have a dependency on view state and you have a limited number of control events, consider using the MVC pattern.

• Consider using the Intercepting Filter pattern to implement the processing steps as pluggable filters when appropriate.

Session Management

When designing a Web application, an efficient and secure session management strategy is important for performance and reliability. You must consider session management factors such as what to store, where to store it, and how long information will be kept.

When designing a session management strategy, consider the following guidelines

• If you have a single web server, require optimum session state performance, and have a relatively limited number of concurrent sessions, use the in-process state store.

• If you have a single web server, your sessions are expensive to rebuild, and you require durability in the event of an ASP.NET restart, use the session state service running on the local web server.

• Use a remote session state service or the SQL Server state store for web farm scenarios.

• Protect your session state communication channel.

• Prefer basic types for session data to reduce serialization costs.

Validation

Designing an effective validation solution is important for the security and reliability of your application. Improper or weak authorization can leave your application vulnerable to cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attack.

When designing a validation strategy, consider following guidelines:

• Identify trust boundaries within Web application layers, and validate all data crossing these boundaries.

• Assume that all client-controlled data is malicious and needs to be validated.

• Design your validation strategy to constrain, reject, and sanitize malicious input.

• Design to validate input for length, range, format, and type.

• Use client side validation for user experience, and server side validation for security.

Presentation Layer Considerations

The presentation layer of your Web application displays the user interface and facilitates user interaction. The design should focus on separation of concerns, where the user interaction logic is decoupled from the user interface components.

When designing the presentation layer, consider following guidelines:

- Consider separating the user interface components from the user interface process components.
- Use client-side validation to improve user experience and responsiveness, and server-side validation for security. Do not rely on just on client-side validation. .
- Use page output caching or fragment caching to cache static pages or parts of pages.
- Use web server controls if you need to compile these controls into an assembly for reuse across applications, or if you need to add additional features to existing server controls.
- Use web user controls if you need to reuse UI fragments on several pages, or if you want to cache specific part of the page.

Business Layer Considerations

When designing the business layer for your Web application, consider how to implement the business logic and long-running workflows. Design business entities that represent the real world data, and use these to pass data between components.

When designing the business layer, consider following guidelines:

• Design a separate business layer that implements the business logic and workflows. This improves maintainability and testability of your application.

• Consider centralizing and re-using common business logic functions.

• Design your business layer to be stateless. This helps to reduce resource contention and increase performance.

• Use a message-based interface for the business layer. This works well with a stateless web application business layer.

• Design transactions for business critical operations.

Data Layer Considerations

- Design a data layer for your Web application to abstract the logic necessary to access the database. Using a separate data layer makes the application easier to configure and maintain. The data layer may also need to access external services using service agents. When designing the data layer, consider following guidelines.
- Design a separate data layer to hide the details of the database from other layers of the application.
- Design entity objects to interact with other layers, and to pass the data between them.
- Design to take advantage of connection pooling to minimize the number of open connections.
- Design an exception handling strategy to handle data access errors, and to propagate exceptions to business layers.
- Consider using batch operations to reduce round trips to the database.

Service Layer Considerations

Consider designing a separate service layer if you plan to deploy your business layer on a remote tier, or if you plan to expose your business logic using a Web service.

When designing the service layer, consider following guidelines:

- If your business layer is on a remote tier, design coarse-grained service methods to minimize the number of client-server interactions, and to provide loose coupling.
- Design the services without assuming a specific client type.
- Design the services to be idempotent

Testing and Testability Considerations

Testability is a measure of how well a system or components allow you to create test criteria and execute tests to determine if the criteria are met. You should consider testability while designing the architecture because it makes it easier to diagnose problems earlier and reduce maintenance cost. To improve testability of your application, you can use logging events, provide monitoring resources, and implement test interfaces.

Consider the following guidelines for testability:

• Clearly define the inputs and outputs of the application or components during the design phase.

• Consider using the Passive View pattern (a variation of the MVP pattern) in the presentation layer, which removes the dependency between the View and the Model.

• Design a separate business layer to implement the business logic and workflows, which improves the testability of your application.

• Design an effective logging strategy, which allows you to detect bugs that might otherwise be difficult to detect. Logging will help you to focus on faulty code when bugs are found. Log files should contain information that can be used to replicate the issues.

• Design loosely coupled components that can be tested individually.

Performance Considerations

- You should identify your performance objectives early in the design phase of a Web application by gathering the non-functional requirements. Response time, throughput, CPU, memory, and disk I/O are few of the

key factors you should consider while designing your application. Consider the following guidelines for performance:

- Ensure the performance requirements are specific, realistic, and flexible.
- Implement caching techniques to improve the performance and scalability of the application.
- Perform batch operations to minimize the round trips across boundaries.
- Reduce the volume of HTML transferred between server and client.
- Avoid unnecessary round trips over the network.

Security Considerations

Security is an important consideration for protecting the integrity and privacy of the data and the resources of your Web application. You should design a security strategy for your Web application that uses tested and proven security solutions; and implement authentication, authorization, and data validation to protect your application from a range of threats. Consider the following guidelines for security:

• Consider the use of authentication at every trust boundary.

• Consider implementing a strong authorization mechanism to restrict resource access and protect business logic.

• Consider the use of input validation and data validation at every trust boundary to mitigate security threats such as cross-site scripting and code-injection.

• Do not rely on only client-side validation. Use server-side validation as well.

• Consider encrypting and digitally signing any sensitive data that is sent across the network.

Deployment Considerations

When deploying a Web application, you should take into account how layer and component location will affect the performance, scalability and security of the application. You may also need to consider design trade-offs. Use either a distributed or a non-distributed deployment approach, depending on the business requirements and infrastructure constraints. Consider the following guidelines for deployment:

• Consider using non-distributed deployment to maximize performance.

• Consider using distributed deployment to achieve better scalability and to allow each layer to be secured separately.

Non-Distributed Deployment

In a non-distributed deployment scenario, all the logically separate layers of the Web application are physically located on the same Web server, except for the database. You must consider how the application will handle multiple concurrent users, and how to secure the layers that reside on the same server. Figure 2 shows this scenario.

Figure 2 - Non-distributed deployment of a Web application.

Consider the following guidelines:

• Consider using non-distributed deployment if your Web application is performance sensitive, because the local calls to other layers provide performance gains.

• Consider designing a component-based interface for your business layer.

• If your business logic runs in the same process, avoid authentication at the business layer.
• Consider using a trusted identity (through the trusted subsystem model) to access the database. This improves the performance and scalability of your application.

• Consider encrypting and digitally signing sensitive data passed between the Web server and database server.

Distributed Deployment

In a distributed deployment, the presentation and business layers of the Web application reside on separate physical tiers, and communicate remotely. You will typically locate your business and data access layers on the same sever. Figure 3 shows this scenario.

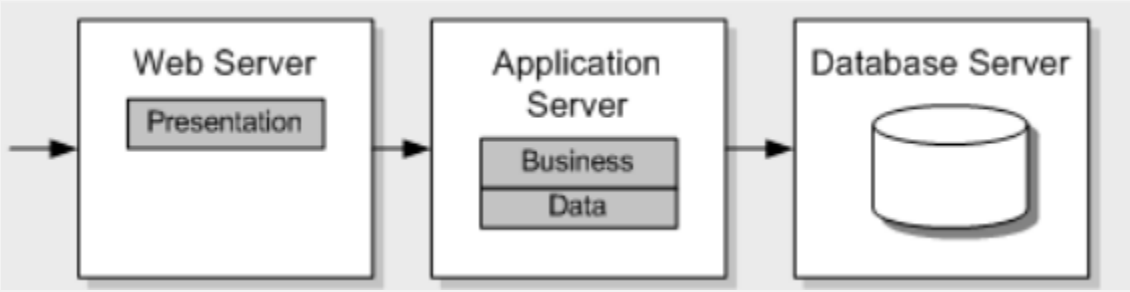

Figure 3 - Distributed deployment of a Web application.

Consider the following guidelines when choosing a distributed deployment:
• Do not physically separate your business logic components unless this is necessary.
• If your security concerns prohibit you from deploying your

business logic on your front-end web server, consider distributed deployment.

• Consider using a message-based interface for your business layer.
• Consider using the TCP protocol with binary encoding to communicate with the business layer for best performance.

• Consider protecting sensitive data passed between different physical tiers.

Load Balancing

When you deploy your Web application on multiple servers, you can use load balancing to distribute requests so that they are handled by different Web servers. This helps to maximize response times, resource utilization, and throughput. Figure 4 shows this scenario.

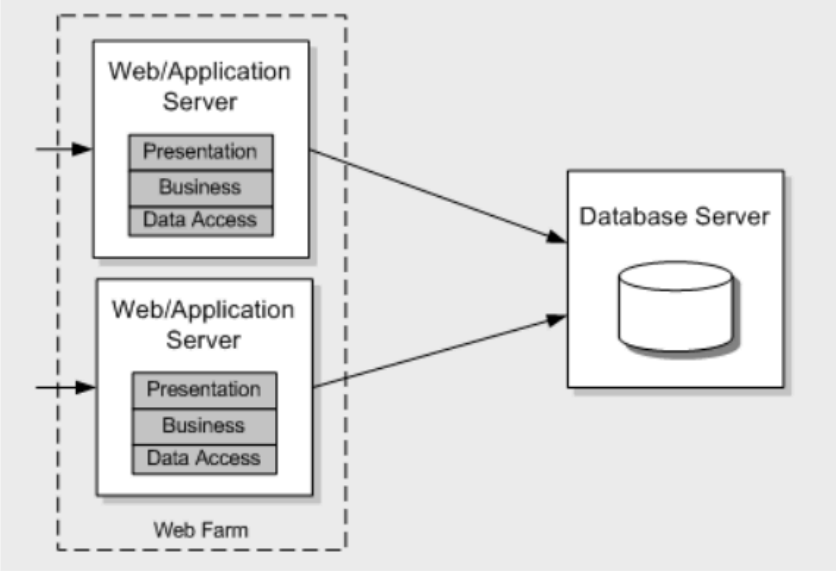

Figure 4 - Load balancing a Web application.

Consider the following guidelines when designing your Web application to use load balancing:

• Avoid server affinity when designing scalable Web applications. Server affinity occurs when all requests from a particular client must be handled by the same server. It usually occurs when you use locally updatable caches, or in-process or local session state stores.

• Consider designing stateless components for your Web application; for example, a Web front end that has no in-process state and no stateful business components.

• Consider using Windows Network Load Balancing (NLB) as a software solution to implement redirection of requests to the servers in an application form.

Web Farm Considerations

A Web farm allows you to scale out you application, which can also minimize the impact of hardware failures. When you add more servers, you can use either a load balancing or a clustering approach. Consider the following guidelines:

• Consider using clustering to minimize the impact of hardware failures.

• Consider partitioning your database across multiple

database servers if your application has high input/output requirements.

• Consider configuring the web farm to route all requests from the same user to the same server to provide affinity where this is required.

• Do not use in-process session management in a web farm when requests from the same user cannot be guaranteed to be routed to the same server. Use an out-of-process state server service or a database server for this scenario.

Pattern Map

Category	Relevant Patterns
Caching	• Cache Dependency • Page Cache
Exception Management	• Exception Shielding
Logging and Instrumentation	• Provider
Navigation	• Model View Presenter • Model View Controller
Page Layout (UI)	• Template View • Composite View • Transform View • Two Step View
Request Processing	• Page Controller • Front Controller • Passive View • Supervising Controller
Service Interface Layer	• Façade • Service Interface

Pattern Descriptions

• **Cache Dependency -** Use external information to determine the state of data stored in a cache.
• **Composite View -** Combine individual views into a composite representation.

• **Exception Shielding -** Filter exception data that should not be exposed to external systems or users.
• **Façade** – Implement a unified interface to a set of operations to provide a simplified reduce coupling between systems.

• **Front Controller -** Consolidate request handling by channeling all requests through a single handler object, which can be modified at runtime with decorators.

• **Model View Controller -** Separate the user interface code into three separate units; Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.

• **Model View Presenter -** Separate request processing into three separate roles, with the View being responsible for handling user input and passing control to a Presenter object.

• **Page Cache -** Improve the response time for dynamic Web pages that are accessed frequently, but change less often and consume a large amount of system resources to construct.

• **Page Controller -** Accept input from the request and handle it for a specific page or action on a Web site.

• **Passive View** – Reduce the view to the absolute minimum by allowing the controller to process user input and maintain the responsibility for updating the view.

• **Provider** – Implement a component that exposes an API that is different from the client API to allow any custom implementation to be seamlessly plugged in.

• **Service Interface** – A programmatic interface that other systems can use to interact with the service**.**

• **Supervising Controller** – A variation of the MVC pattern in which the controller handles complex logic, in particular coordinating between views, but the view is responsible for simple view-specific logic.

• **Template View -** Implement a common template view, and derive or construct views using this template view.

• **Transform View** – Transform the data passed to the presentation tier into HTML to be displayed on UI.

• **Two Step View** – Transform the model data into a logical presentation without any specific formatting, and then

convert that logical presentation into the actual formatting required.

Presentation Layer

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

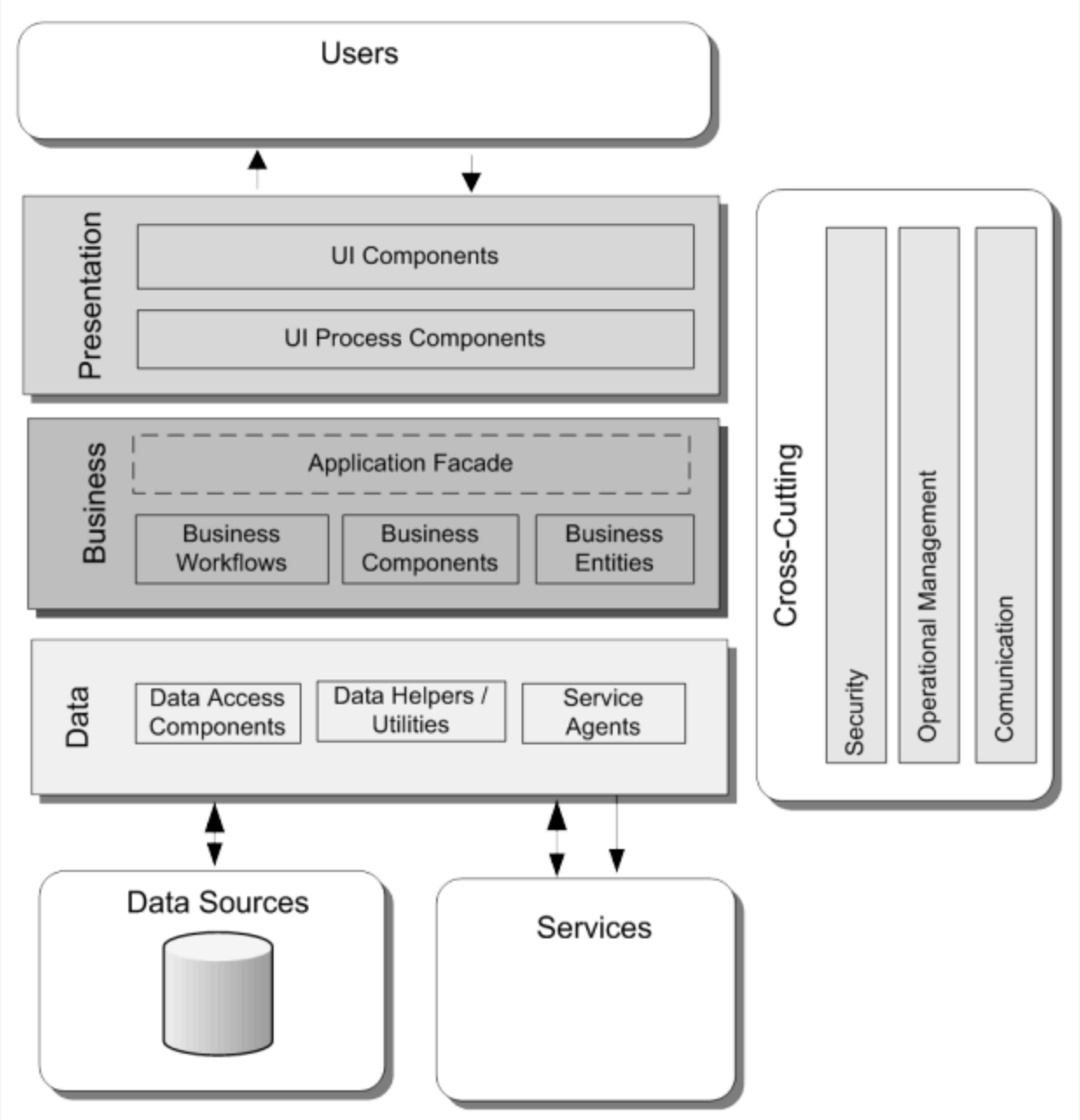

Figure 1 - A typical application showing the presentation layer and the components it may contain.

Presentation Layer Components

• ***User interface (UI) components.***
User interface components provide a way for users to interact with the application. They render and format data for users. They also acquire and validate data input by the user.
• ***User process components.***

User process components synchronize and orchestrate user interactions. Separate user process components may be useful if you have a complicated user interface. Implementing common user interaction patterns as separate user process components allows you to reuse them in multiple user interfaces.

Approach

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

1. Determine how you will **present data**. Choose the data format for your presentation layer and decide how you will present the data in your User Interface (UI).

2. Determine your **data validation strategy**. Use data validation techniques to protect your system from untrusted input.

3. Determine your **business logic strategy**. Factor out your business logic to decouple it from your presentation layer code.

4. Determine your **strategy for communication** with other layers. If your application has multiple layers, such as a data access layer and a business layer, determine a strategy for communication between your presentation layer and other layers.

Design Considerations

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

• **Use the relevant patterns.** Review the presentation layer patterns for proven solutions to common presentation problems.

• **Design for separation of concerns.** Use dedicated UI components that focus on rendering and display. Use dedicated presentation entities to manage the data required to

present your views. Use dedicated UI process components to manage the processing of user interaction.

• **Consider human interface guidelines.** Review your organization's guidelines for user interface design. Review established user interface guidelines based upon the client type and technologies that you have chosen.

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

Presentation Layer Frame

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

Category	Common issues
Caching	• Caching volatile data.
	• Caching unencrypted sensitive data.
	• Incorrect choice of caching store.

	• Failing to choose a suitable caching mechanism for use in a Web farm.
	• Assuming that data will still be available in the cache – it may have expired and been removed.
Composition	• Failing to consider use of patterns and libraries that support dynamic layout and injection of views and presentation at runtime.
	• Using presentation components that have dependencies on support classes and services instead of considering patterns that support run-time dependency injection.
	• Failing to use the Publish/Subscribe pattern to support events between components.
	• Failing to properly decouple the application as separate modules that can be added easily.
Exception Management	• Failing to catch unhandled exceptions.
	• Failing to clean up resources and state after an exception occurs.
	• Revealing sensitive information to the end user.
	• Using exceptions to implement application logic.
	• Catching exceptions you do not handle.
	• Using custom exceptions when not necessary.
Input	• Failing to design for intuitive use, or implementing overcomplex interfaces.
	• Failing to design for accessibility.
	• Failing to design for different screen sizes and resolutions.
	• Failing to design for different device and input types, such as mobile devices, touch-screen, and pen and ink enabled devices.

Layout	• Using an inappropriate layout style for Web pages.
	• Implementing an overly-complex layout.
	• Failing to choose appropriate layout components and technologies.
	• Failing to adhere to accessibility and usability guidelines and standards.
	• Implementing an inappropriate workflow interface.
	• Failing to support localization and globalization.
Navigation	• Inconsistent navigation.
	• Duplication of logic to handle navigation events.
	• Using hard-coded navigation.
	• Failing to manage state with wizard navigation.
Presentation Entities	• Defining entities that are not necessary.
	• Failing to implement serialization when necessary.
Request Processing	• Blocking the user interface during long-running requests.
	• Mixing processing and rendering logic.
	• Choosing an inappropriate request-handling pattern.
User Experience	• Displaying unhelpful error messages.
	• Lack of responsiveness.
	• Over-complex user interfaces.
	• Lack of user personalization.
	• Lack of user empowerment.
	• Designing inefficient user interfaces.
UI Components	• Creating custom components that are not necessary.

	• Failing to maintain state in the MVC pattern.
	• Choosing inappropriate UI components.
UI Process Components	• Implementing UI process components when not necessary.
	• Implementing the wrong design patterns.
	• Mixing business logic with UI process logic.
	• Mixing rendering logic with UI process logic.
Validation	• Failing to validate all input.
	• Relying only on client-side input validation. You must always
	validate input on the server or in the business layer as well.
	• Failing to correctly handle validation errors.
	• Not identifying business rules that are appropriate for
	validation.
	• Failing to log validation failures.

Data Layer

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

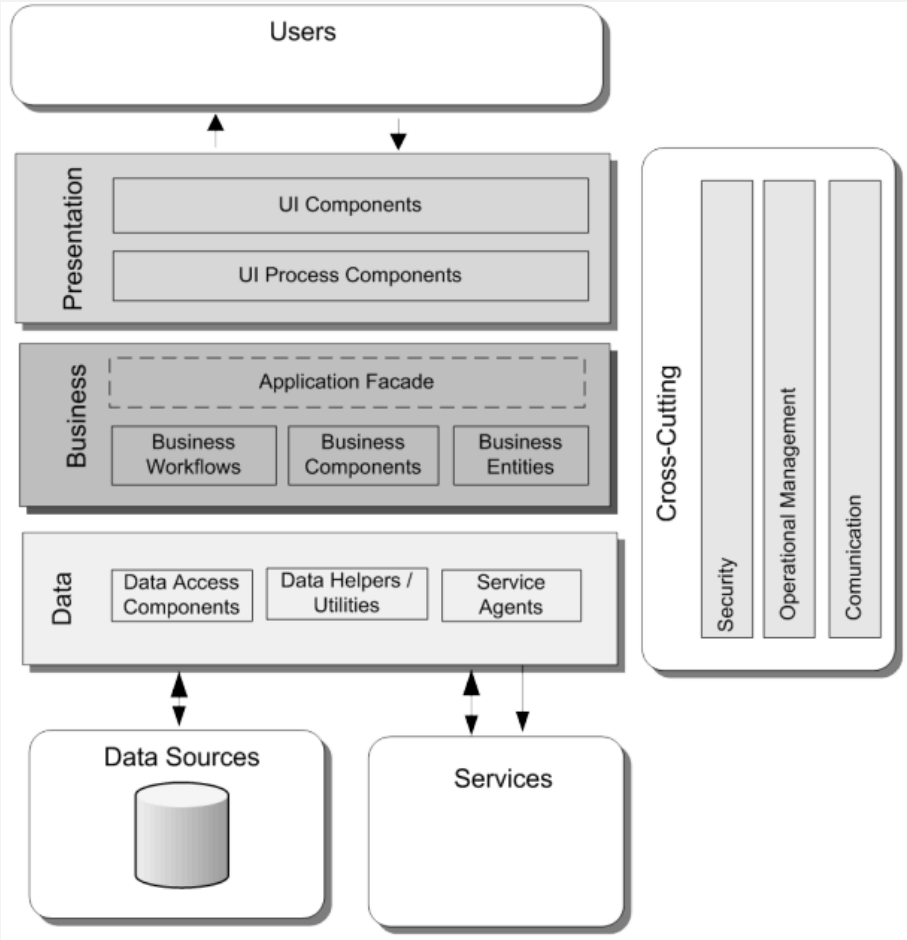

Figure 1 - A typical application showing the data layer and the components it may contain.

Data Layer Components

• ***Data access logic components.***

Data access components abstract the logic necessary to access your underlying data stores. Doing so centralizes the data access functionality, which makes the application easier to configure and maintain.

• ***Data Helpers / Utilities.***

Helper functions and utilities assist in data manipulation, data transformation, and data access within the layer. They consist of specialized libraries and/or custom routines especially designed to maximize data access performance and reduce the development requirements of the logic components and the service agent parts of the layer.

• ***Service agents***.

When a business component must use functionality exposed by an external service, you may need to create code that manages the semantics of communicating with that service. Service agents isolate your application from the idiosyncrasies of calling diverse services, and can provide additional services such as basic mapping between the format of the data exposed by the service and the format your application requires.

Approach

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

1. **Create an overall design for your data access layer:**

a. Identify your data source requirements
b. Determine your data access approach
c. Choose how to map data structures to the data source
d. Determine how to connect to the data source
e. Determine strategies for handling data source errors.

2. **Design your data access components**:

a. Enumerate the data sources that you will access

b. Decide on the method of access for each data source
c. Determine whether helper components are required or desirable to simplify data
access component development and maintenance

d. Determine relevant design patterns. For example, consider using the Table Data
Gateway, Query Object, Repository, and other patterns.

3. **Design your data helper components:**

a. Identify functionality that could be moved out of the data access components and
centralized for reuse

b. Research available helper component libraries
c. Consider custom helper components for common problems such as connection

strings, data source authentication, monitoring, and exception processing

d. Consider implementing routines for data access monitoring and testing in your
helper components

e. Consider the setup and implementation of logging for your helper components.

4. **Design your service agents:**

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

Design Guidelines

The following design guidelines provide information about different aspects of the data access layer that you should

consider. Follow these guidelines to ensure that your data access layer meets the requirements of your application, performs efficiently and securely, and is easy to maintain and extend as business requirements change.

• **Choose the data access technology.**

The choice of an appropriate data access technology
will depend on the type of data you are dealing with, and how you want to manipulate the data within the application. Certain technologies are better suited for specific scenarios. The following sections of this guide discuss these options and enumerate the benefits and drawbacks of each data access technology.

• **Use abstraction to implement a loosely coupled interface to the data access layer**.

This can be accomplished by defining interface components, such as a gateway with well-known inputs and outputs, which translate requests into a format understood by components within the layer. In addition, you can use interface types or abstract base classes to define a shared abstraction that must be implemented by interface components.

• **Consider consolidating data structures.** If you are dealing with table-based entities in your data access layer, consider using Data Transfer Objects (DTOs) to help you

organize the data into unified structures. In addition, DTOs encourage coarse-grained operations while providing a structure that is designed to move data across different boundary layers.

• **Encapsulate data access functionality within the data access layer**. The data access layer hides the details of data source access. It is responsible for managing connections, generating queries, and mapping application entities to data source structures. Consumers of the data access layer interact through abstract interfaces using application entities such as custom objects, DataSets, DataReaders, and XML documents. Other application layers
that access the data access layer will manipulate this data in more complex ways to implement the functionality of the application. Separating concerns in this way assists in application development and maintenance.

• **Decide how to map application entities to data source structures.** The type of entity you use in your application is the main factor in deciding how to map those entities to data source structures.

Decide how you will manage connections. As a rule, the data access layer should create and manage all connections to all data sources required by the application. You must choose an appropriate method for storing and protecting connection

information that conforms to application and security requirements.

• **Determine how you will handle data exceptions.** The data access layer should catch and (at least initially) handle all exceptions associated with data sources and CRUD operations. Exceptions concerning the data itself, and data source access and timeout errors, should be handled in this layer and passed to other layers only if the failures affect application responsiveness or functionality.

• **Consider security risks**. The data access layer should protect against attacks that try to steal or corrupt data, and protect the mechanisms used to gain access to the data source. It should also use the "least privilege" design approach to restrict privileges to only those needed to perform the operations required by the application. If the data source itself has the ability to limit privileges, security should be considered and implemented in the data access layer as well as in the source.

• **Reduce round trips**. Consider batching commands into a single database operation.

• **Consider performance and scalability objectives**. Scalability and performance objectives for the data access layer should be taken into account during design. For example, when designing an Internet-based merchant

application, data layer performance is likely to be a bottleneck for the application. When data layer performance is critical, use profiling to
understand and then limit expensive data operations.

Data Layer Frame

Category	Common Issues
BLOB	• Improperly storing BLOBs in the database instead of the file system.
	• Using an incorrect type for BLOB data in database.
	• Searching and manipulating BLOB data.
Batching	• Failing to use batching to reduce database round-trips .
	• Holding onto locks for excessive periods when batching.
	• Failing to consider a strategy for reducing database round-trips with batching.
Connections	• Improper configuration of connection pooling.
	• Failing to handle connection timeouts and disconnections.
	• Performing transactions that span multiple connections.
	• Holding connections open for excessive periods.
	• Using individual identities instead of a trusted subsystem to access the database.
Data Format	• Choosing the wrong data format.
	• Failing to consider serialization requirements.
	• Not Mapping objects to a relational data store.
Exception Management	• Not handling data access exceptions.
	• Failing to shield database exceptions from the original caller.
	• Failing to log critical exceptions.

Queries	• Using string concatenation to build queries.
	• Mixing queries with business logic.
	• Not optimizing the database for query execution.
Stored Procedures	• Not passing parameters to stored procedures correctly.
	• Implementing business logic in stored procedures.
	• Not considering how dynamic SQL in stored procedures can impact performance, security, and maintainability.
Transactions	• Using the incorrect isolation level.
	• Using exclusive locks, which can cause contention and deadlocks.
	• Allowing long-running transactions to blocking access to data.
Validation	• Failing to perform data type validation against data fields.
	• Not handling NULL values.
	• Not filtering for invalid characters.
XML	• Not considering how to handle extremely large XML data sets.
	• Not choosing the appropriate technology for XML to relational database interaction.
	• Failure to set up proper indexes on applications that do heavy querying with XML
	• Failing to validate XML inputs using schemas.

Technology Considerations

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

designing and the requirements of
that application:

- If you require basic support for queries and parameters, consider using ADO.NET objects
directly.
- If you require support for more complex data-access scenarios, or need to simplify your
data access code, consider using the Enterprise Library Data Access Application Block.

- If you are building a data-driven Web application with pages based on the data model of the
underlying database, consider using ASP.NET Dynamic Data.

- If you want to manipulate XML-formatted data, consider using the classes in the System.Xml
namespace and its subsidiary namespaces.

- If you are using ASP.NET to create user interfaces, consider using a DataReader to access
data to maximize rendering performance. DataReaders are ideal for read-only, forward-only
operations in which each row is processed quickly.

- If you are accessing Microsoft SQL Server, consider using classes in the ADO.NET SqlClient
namespace to maximize performance.

• If you are accessing Microsoft SQL Server 2008, consider using a FILESTREAM for greater
flexibility in the storage and access of BLOB data.

• If you are designing an object oriented business layer based on the Domain Model pattern,
consider using the ADO.NET Entity Framework.

Business Logic Vs Application Logic

We define business logic as the subset of the application's code which manipulates business objects. If we relate this to the MVC approach described before, business logic manipulation takes place in the Model layer. The application logic, on the other hand, is the subset of the code which manipulates the inner part of the application : interface, protocols, abstract objects, etc.

Thus, the application logic is irrelevant to both the end user and the people writing the specifications for the application. Separating the business logic from the application logic is a good coding practice, and the MVC pattern was designed to facilitate that. However, this can be difficult to achieve.

The Model-View-Controller pattern

At its simplest, a web application answers to the following paradigm : provide to its users a human-computer interface to perform operations on business-related data. Therefore, the development of a web application can be made in three steps :

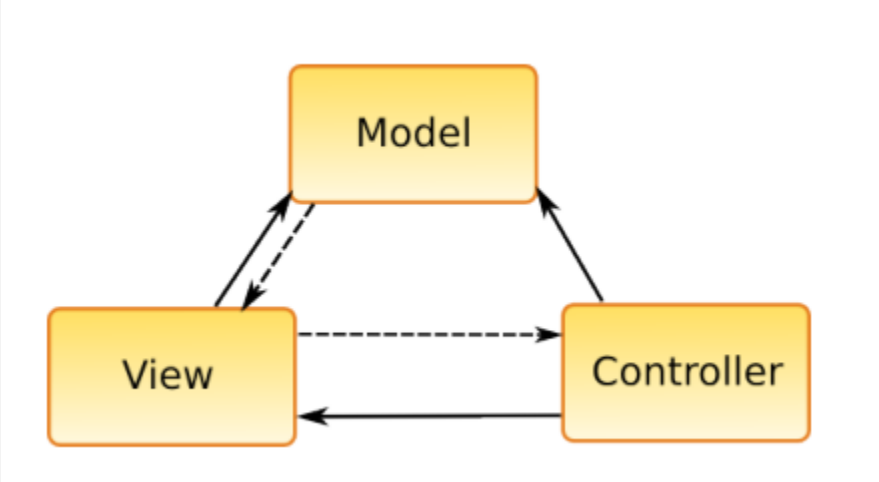

Fig: Model View Controller Pattern

- at the lower level, building a business logic handler, which would process and store the business data, either internally in a database, or externally by transferring it through a web service,
- at the higher level, building the human-computer interface,
- finally, ”wiring” these layers together, by dealing with incoming requests, transferring the business data to the view, and reacting to user inputs on the data.

It is quite common that these concerns be mixed. This leads to problems for managing business data, which is embedded within different aspects of the code. For example, the business data is managed at the low level, like database access, but also the highest level, when it comes to making that data readable to the end user (i.e. processing HTML pages specific to that data). Without a common policy to organize the business code, any developer in the project could implement business data manipulation at any layer. Therefore, any change would request a thorough check of the code. Based on this fact, a programming pattern rose in the eighties [bur92], called Model-View-Controller2. When using a MVC pattern, the development of the application is formalized, and the code is divided into several tiers.

- the *model* embeds the application and business logic. It is the ”smart” part of the application,
- the *view* deals with the interface. It is made of ”dumb” templates designed to embed specific information into generic HTML pages,
- the *controller* ”wires” the elements together, by dealing with incoming requests, processing the data and directing it to the view.

Model

The model is an abstracted view of the application. It is a stateful representation of the relevant data with which the application interacts. When the application is mapped with a database, the model is the part which interacts with it directly. The model deals directly with business data : as it handles the data access tier, all business elements transit through it, and are stored in it during a session. In a classical Java approach, the model consists of an Object-Relational Mapping entity, which makes the contents of the database accessible as Java objects. For example, let's consider one database table *Customer* and its child table *Account*, joined by a one-to-many relationship (one customer has many accounts). One row of the Customer table would be mapped in Java as a Customer object, with a function getAccounts() which would return all the joined rows in the Account table, i.e. all the accounts the customer is subscribed to.

The manipulation of business objects takes place in the Model. For example, if the programmer of our application needs a function to select all the customers who have been subscribed for more than one year, he should write it in the Model.

In our case, the Model is also responsible for handling the communication to external entities via Webservices, and all the XML parsing and generating involved.

View

If done correctly, the view should consist of HTML pages with atomic requests to the controller, and simple processing (like loops). With a Java approach, it would take concretely the form of JSP (JavaServer Pages).

Example Here is the display code for the display of the maximum power :

```
<h1>List of subscriptions with maximum power :</h1>
<table>
<tr>
<td>Meter ID</td>
<td>Max Power</td>
</tr>
<%for ( i =0;i<subscription . length ; i++) { %>
<tr>
<td><%=request . getParameter (meterID [ i])%></td>
<td><%=request . getParameter (maxPower[ i])%> W
</td>
</tr>
<% } %>
</table>
```

Since this tier doesn't involve business logic directly, its further description is out of scope of this thesis.

Controller

The controller maps the model with the view. It responds to user actions and redirect him to the correct screen. While some of these actions are related to the program itself (like login, interface choice, etc...) most of them also operate on the business logic. Therefore, while the Model was dealing with business objects, the Controller handles the business rules : different business data put in the program by the user will yield different actions. Such data can be trivial (like, when one input integer is over a threshold, request one extra parameter for input) or very intricate, depending on real-life business considerations.

Therefore, like the Model handles the business elements, the controller deals with the business rules. It checks if the values are compatible with the business pattern and directs it to the appropriate action.

The Controller is also the tier in contact with the HTTP layer. Java Enterprise Edition provides the HttpServlet Bean, which is an abstraction of HTTP communication. Therefore, the business logic is separated from the communication matters.

Implementation using Apache Struts

A lot of frameworks are available. For this project, Apache Struts was used. Apache Struts is the leading MVC framework for Java Enterprise Edition serverside development. Whenever the user requests for something, the request is handled by the Struts Action servlet. When the Action servlet receives a request, it intercepts the URL and assigns the request handling to the Action class according to the Struts Configuration file (struts-config.xml). To organize our program appropriately, we separate it into several classes that inherit from the Action class. Therefore, in our program, the Model consists of our Model XML file, which contains the business objects, and the Action classes, which contains the business logic related to these objects.

Example In this Action file, we first check that the subscription is not already performed before launching it. This is just given for illustration purposes and we do not guarantee accuracy, nor will we explain all the details :

```
import web . http . s e r v l e t ;
import apache . s t r u t s . FwkAction ; // import the s t r u t s framework
import apache . XMLparser . FwkElement ; /* import the framework
```

```
to read the model XML f i l e */
public class SubscribeAction {
public ActionForward executeAction ( FwkAction action ,
XMLModel model , HttpServletRequest request ,
HttpServletResponse response ) {
i f ( model . getValueOf (
"model/ currentCustomer /meter/@id" )
. equals ( model . getValueOf (
"model/ request / content / meter id " )) {
forward=" same meter error " ;
} else {
/* perform s u b s c r i p t i o n o p e r a t i o n s
and send w e b s e r v i c e */
forward=" s u b s c r i p t i o n s e n t " ;
}
return action . findForward ( forward ) ;
}
}
```

There, according to the test of the equality of two elements in our model XML, the user is redirected to 2 different actions. To figure out which actions are directed to which views, we need to look at the struts-config.xml file.

Here is a sample :

```
<?xml version=" 1.0 " encoding="UTF8"?>
<struts −c o n f i g>
```

<action path=”/ subscribe ”>
<forward name=” same meter error ”
path=” e r r o r s /sameMeterError . jsp ” />
<forward name=” s u b s c r i p t i o n s e n t ”
path=” waiting /waitForAnswer . jsp ” />
</ action>
<!−−*more a c t i o n s* −−>
</ struts −c o n f i g>