**Introduction to Web Architecture**

Table of Contents

[Web Architecture Layers 2](#_Toc100478701)

[Client-Server Architecture 3](#_Toc100478702)

[Thin and Thick Clients 3](#_Toc100478703)

[Tiered Architecture 4](#_Toc100478704)

[1-Tier Architecture 4](#_Toc100478705)

[2-Tier Architecture 5](#_Toc100478706)

[3-Tier Architecture 6](#_Toc100478707)

[n-Tier Architecture 6](#_Toc100478708)

[MVC Architecture 7](#_Toc100478709)

## Web Architecture Layers

**Software architecture** is a view of the system that includes the system’s major components, their behaviour and how they interact and coordinate to achieve the system’s goals.

Web applications basically have three layers:

1. **Presentation Layer** – This layer deals with the user interface and user interactions. This works on the **Client** or **Web Browser**.
2. **Application Layer** – Also called the **Business Logic Layer**, this layer does the main work. It interprets user actions made on the presentation layer, communicates with the database layer and causes the presentation layer to update with new information from the database layer. This works on the **Application Server**.
3. **Database Layer** – This simply deals with the database. It contains methods to communicate with the database. This works on the **Database Server**.

## Client-Server Architecture

The simplest possible software architecture is the **Client-Server Architecture**. This is a basic request-response architecture, with the client sending requests over the network which the server responds to. If we consider the layers, the presentation layer works on the client side while the application and database layers work on the server side.

For example, consider an ATM client. On the client side, the presentation layer allows the user to enter their credentials. This goes to the server, where the application layer works with the database layer to verify the credentials. Once the credentials are verified, the application layer causes the presentation layer on the client side to update.

### Thin and Thick Clients

We can have two types of clients, thin clients and thick clients. **Thin clients** are ones that do not perform any further processing on the data they get from the server. For example, an ATM machine that simply shows the total balance of the user upon receiving the information from the server is a thin client. **Thick clients** are ones that do perform further processing on the data. For example, an ATM machine that can break down the total balance data into month-wise sections is a thick client.

## Tiered Architecture

Under the **tiered architecture**, we can have 1-tier architectures, 2-tier architectures, 3-tier architectures and n-tier architectures.

### 1-Tier Architecture

In the **1-Tier Architecture**, all three layers, the presentation layer, the application layer and the database layer, are on the **same machine**. Such an architecture is not used in industry-grade applications, since those require communication with a centralized server.

* Easy to implement and optimize performance.
* Does not have compatibility or context switching issues.
* Cheaper to deploy and maintain.
* Does not support remote or distributed access for data resources.
* Monolithic manner of code (all the code is in the same package) makes things difficult to maintain, since there is a single source of failure.
* The central mainframe is expensive.

### 2-Tier Architecture

Under the **2-Tier Architecture**, we have either the presentation layer and application layer on the same machine, or the application layer and the database layer on the same machine, typically the latter. The other remaining layer is on a different machine.

* Applications can be easily developed due to simplicity.
* Maximum user satisfaction is gained with accurate and fast prototyping of applications through robust tools.
* Since this contains static business rules, it is more applicable for homogenous environments.
* Database server and business logic are physically close, which offers higher performance.
* Heterogenous environments or business environments with rapidly changing rules and regulations are not suitable, since the database server has to handle the business logic which slows down the database performance.
* There are problems with thick clients.
* There is a lack of scalability, since only a limited number of users can be supported.
* The application layer depends on the database layer, which creates issues if we need to re-design the database layer.

### 3-Tier Architecture

Under the **3-Tier Architecture**, we have all three layers on separate machines, even though the application layer and the database layer are both on the server side.

* Improved scalability
* Enhanced re-use
* Improved data integrity
* Enhanced security
* Redundant server availability, giving multiple points of failure
* Complexity of communication
* Difficulty maintaining coupling
* Increased development time

### n-Tier Architecture

Under the **n-Tier Architecture**, we have multiple components (i.e. machines) working on each layer. Otherwise, it is the same as the 3-Tier Architecture. For example, we could have one machine using an SQL server running on the database layer and another machine using a MongoDB server. The multiple components provide redundancy, but also increase complexity.

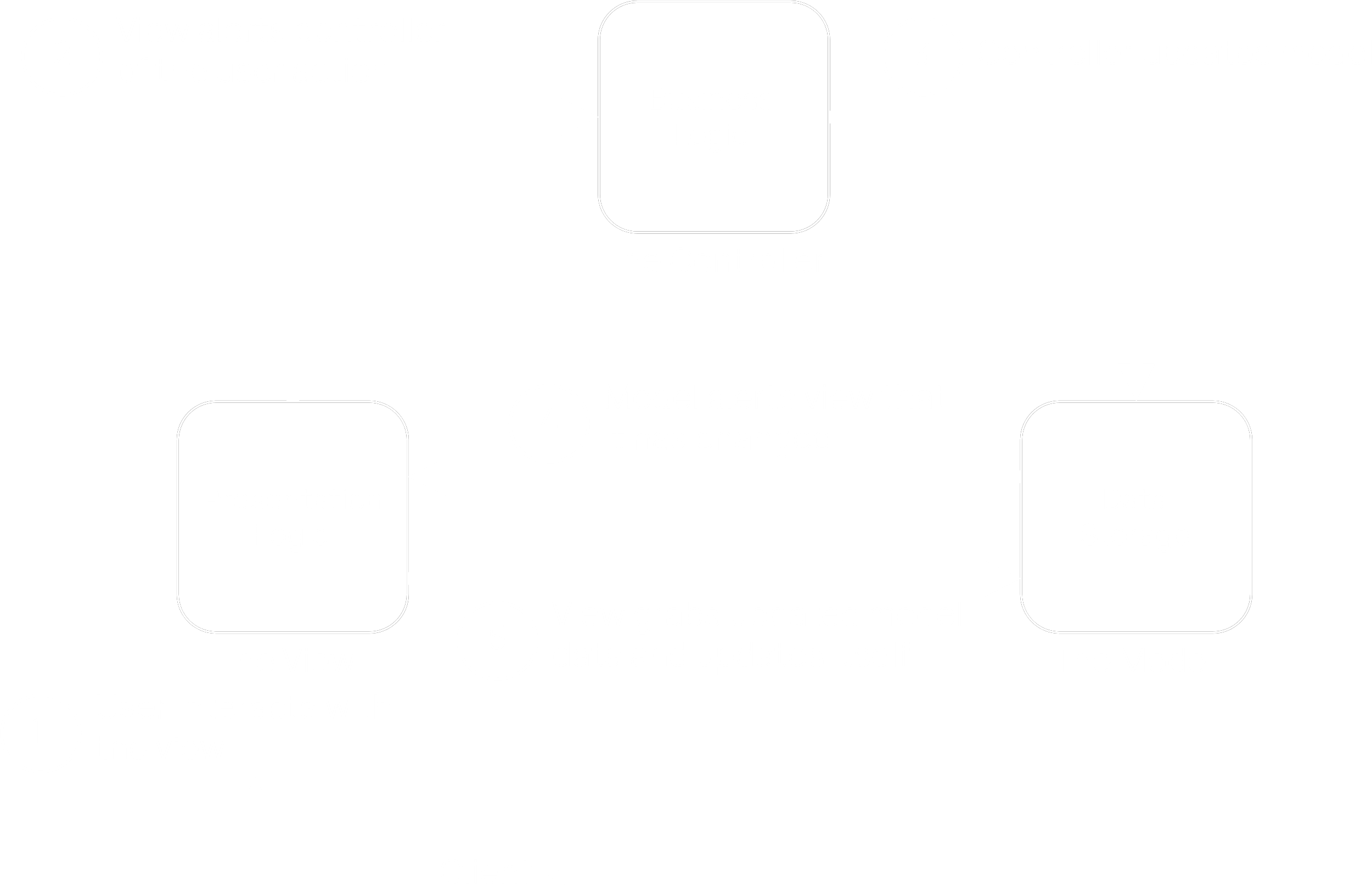
## MVC Architecture

MVC stands for **Model, View, Controller**. Under the **MVC architecture**, the Model deals with the Data Storage Layer, the View deals with the Presentation Layer and the Controller deals with the Business Logic.

The **model** deals with the database as well as any logic associated with the database, such as PL/SQL functions. Depending on the logic in the controller, the required data is extracted from the database and stored in the model.

The **view** retrieves user interactions and presents the information to the user.

The **controller** interprets user actions made in the view. It interprets the actions of the user so that it can command the model to retrieve data, which it sends back to the view to update it accordingly.



* Easy code maintenance
* Components can be tested individually.
* Easier support for new types of clients, since we just need to modify the presentation layer.
* Different components can be developed parallelly.
* The Front-Controller design pattern can be used, which allows processing of web application requests through a single controller.
* It works well for web apps which are supported by large teams of web designers and developers.
* Provides clean Separation of Concerns (SoC)
* The framework can be complex.
* Data management becomes complex and inefficient.
* To take advantage of parallel development, we need multiple programmers.
* Knowledge of multiple technologies is required.
* There is high code maintenance.