



Multi-Tier Architectures and GUIs

What is a Multi-Tier Architecture?

Definition

- Define Multi-Tier Architecture as a software architecture pattern where the application is divided into logical layers or tiers, typically a Data Tier, Logic Tier, and Presentation Tier.
- Primary goal: separation of concern

Key Components

- Data Tier: Database and data storage components.
- Logic Tier: Business logic, application server.
- Presentation Tier: User interface, frontend components.

Properties

1. **Stack-like Hierarchy:** The layers in a multi-tier system often work as a sequence of calls where each layer communicates only with its adjacent layers.
2. **Interchangeable Layers:** Ideally, you should be able to swap out one layer for another (e.g., change the database) without affecting the other layers.
3. **Universality:** Multi-tier architecture is a broader architectural pattern that can be implemented in any aspect of software design and is not limited to UI-centric applications.

Multi-tier ≠ UI centric

Certainly, multi-tier architectures are not limited to UI-centric applications. They can be employed in a variety of contexts to separate concerns and simplify system design. Here are some examples:

1. Backend Services in a Microservices Architecture
 - **Data Layer:** Databases, caching systems, and storage services.
 - **Application Layer:** Individual microservices handling specific business logic.
 - **API Gateway Layer:** A unified API Gateway that routes requests to appropriate microservices.
 - **Edge Layer:** Load balancers and firewalls for handling incoming client requests.
2. Financial Transaction Systems
 - **Transaction Capture Layer:** First layer that captures the basic transaction details from various channels (e.g., ATM, Internet Banking, etc.)
 - **Processing Layer:** The main logic for handling transactions, verifying user accounts, etc.
 - **Ledger Layer:** Where the final transaction is recorded, often in a secure, auditable database.
 - **Reporting Layer:** Generates reports and logs for auditing and compliance.
3. Internet of Things (IoT) Systems
 - **Device Layer:** IoT devices that collect data.
 - **Edge Computing Layer:** Intermediate hardware that can do preliminary data processing (i.e. processing video signals, multiplexing/syncing video and audio).
 - **Data Aggregation Layer:** Where data from multiple devices are aggregated, often in the cloud.
 - **Application Layer:** Where the bulk of the business logic resides, including rules for alerts, data analytics, etc.

Advantages

- Scalability: Each tier can scale independently.
- Maintainability: Easier to update or modify one tier without affecting the others.
- Separation of Concerns: Each tier focuses on a specific aspect of the application.

Disadvantages

- Complexity: More tiers can make the system more complex to understand and manage.
- Latency: Multiple tiers can introduce latency due to communication overhead.
- Risk of Over-Engineering: Not every project needs a multi-tier architecture.

GUI Architectures

- While multi-tier architecture is often thought of in the context of backend services, it is highly relevant in the world of GUI-based applications as well.
- MVC, MVP, and MVVM are essentially specific forms of a multi-tier architecture focused on user interfaces.

MVC (Model-View-Controller)

- MVC stands for Model-View-Controller. It was created by Trygve Reenskaug while working on Smalltalk-79. In his design, a Model represents the knowledge in the system. A View is a visual representation of the Model, retrieving data from the Model to display to the user and passing requests back and forth between the user and the Model. A Controller is an organizational part of the user interface that lays out and coordinates multiple Views on the screen, and which receives user input and sends the appropriate messages to its underlying Views.
- **Model:** Responsible for data and business rules.
- **View:** Displays the data, listens for user interaction, and delegates user inputs to the controller.
- **Controller:** Accepts user input, processes it (with possible updates to the model), and returns the output display to the view.
- **Communication:** Controller updates the Model, View updates itself from the Model and sends messages to the Controller.
- **Best Suited For:** Web applications where server-side logic plays a critical role.
- **Examples:** Spring Framework (Java), Django (Python), Ruby on Rails (Ruby).

MVP (Model-View-Presenter)

- **Model:** Holds the data and business rules.
- **View:** Displays the data and routes user commands to the Presenter to act upon that data.
- **Presenter:** Acts upon the Model and the View; it retrieves data from the Model, manipulates it, and returns it to the View.
- **Communication:** Presenter updates the View, View updates the Presenter, Presenter updates the Model.
- **Best Suited For:** Applications requiring complex user interfaces, often seen in enterprise-level desktop or mobile applications.
- **Examples:** GWT (Java), Moxy (Android), Windows Forms (C#).

MVVM (Model-View-ViewModel)

- **Model:** Contains data and business rules.
- **View:** Displays data and has mechanisms to communicate user actions to the ViewModel.

- **ViewModel:** Holds the presentational logic and state; updates the Model based on View interaction, and the View updates itself based on ViewModel changes.
- **Communication:** ViewModel updates the View via Data Binding, View updates ViewModel through Command Pattern.
- **Best Suited For:** Applications where UI and UX are a priority, often used in modern web and mobile applications.
- **Examples:** Angular (JavaScript), React + Redux (JavaScript), Xamarin (C#).

Differences

- **User Interaction:** In MVC, the Controller handles user interaction. In MVP, the Presenter does. In MVVM, ViewModel handles data-binding to update the View.
- **Testability:** MVP is generally considered easier to test than MVC because the Presenter relies on the View interface. MVVM also promotes testability.
- **Data Binding:** MVVM relies heavily on data-binding, which is generally not a feature in MVC or MVP out-of-the-box.
- **Complexity:** MVP and MVVM can often require more boilerplate code than MVC but offer advantages in testability and separation of concerns.
- **Framework Support:** MVC is often used in web frameworks, MVP is common in desktop applications, and MVVM is popular in frameworks that have robust data-binding support like Angular.

Summary and Key Takeaways

- Multi-tier architectures divide an application into logical tiers for better separation of concerns, scalability, and maintainability.
- Various GUI architectures like MVC, MVVM, and MVP can fit into the Presentation tier of a multi-tier setup.