Quality Mobile Applications

Exploring Mobile Development: Architectures and Frameworks

Introduction

- World of mobile applications is **competitive**.
- So, it is crucial to create software that is not only **cost-effective** but also of **superior quality**.
- A critical factor in achieving this is the **selection of an appropriate architecture**.
- The right architectural choice ensures the
  - application's long-term quality,
  - facilitates rapid development, and
  - allows for frequent updates to keep users engaged and satisfied.

Introduction

- When developing new applications, **speed** and **efficiency** are paramount.
- **Rapid development** cycles help in **reducing costs**, while **regular updates keep the user base interested and loyal**.
- Therefore, the **architecture of your mobile application plays a vital role** in ensuring its **quality** and **adaptability** over time.
- In this context, it's important to ask:
  - Is your program testable,
  - maintainable, and
  - scalable?

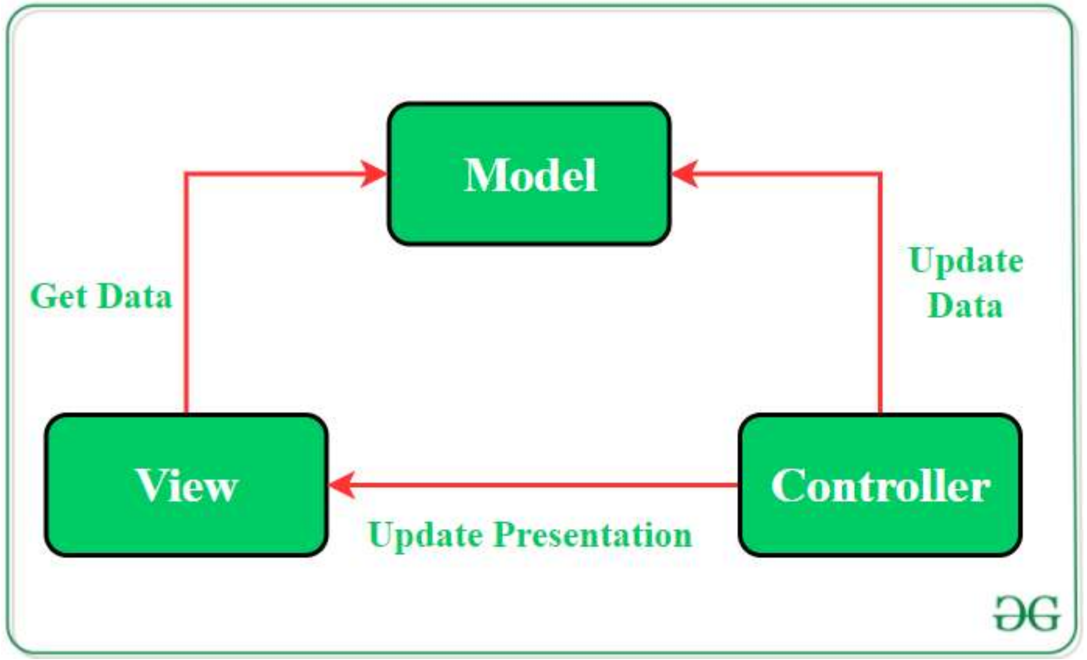
  These attributes are essential for sustaining the application's quality, managing ongoing updates, and accommodating growth.

Introduction

- In software development, applications are continuously evolving.
- Thus, they are becoming more complex as the lines of code increases.
- This increasing complexity poses significant challenges in terms of **maintenance** and **reusability**.
- **Software architecture** provides a **clear description** and **demonstration** of the software's **structure** and **content**.
- **Software architecture** outlines the relationships between the static and dynamic components of an application, ensuring a cohesive and understandable structure.

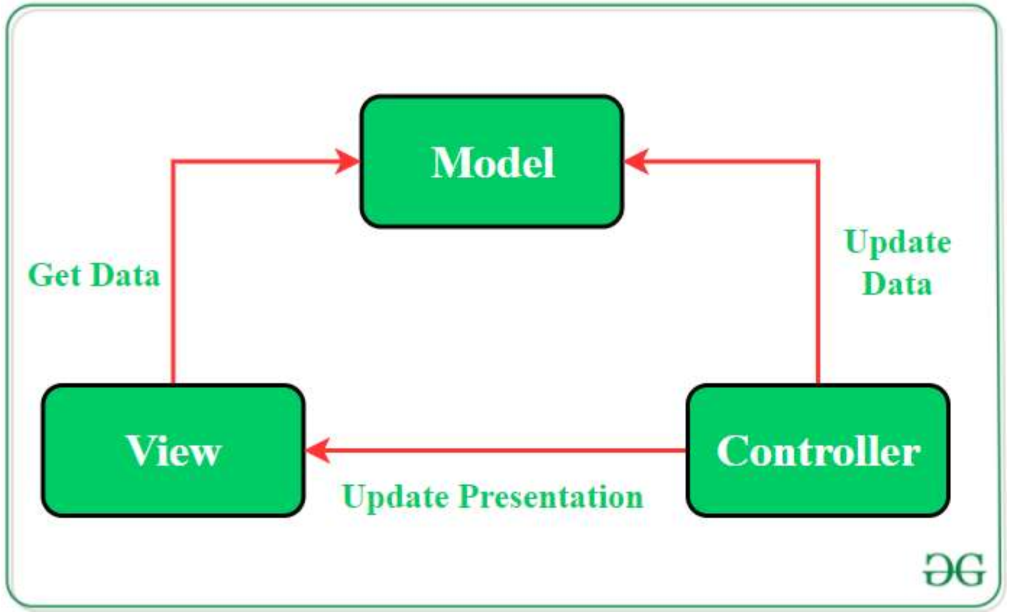
Model-View-Controller (**MVC)**

- **What is it:**
  - MVC is a design pattern that was presented in 1978 by Trygve Reenskaug.
  - It is one of the oldest and most widely adopted architectural patterns in software development, known for its ability to separate the concerns of an application into three interconnected components.
  - This separation makes it easier to manage and scale applications.
- **Goals of MVC**
  - The goal is to separate the business logic from the presentation logic.
  - This separation allows developers to work on the application's business logic without worrying about its presentation layer, and vice versa.
  - It enhances modularity and promotes code reuse.



Model

1. **Role**:
    1. The Model represents the application's data and business logic. It is responsible for managing the data, logic, and rules of the application.
    2. Or, we can say, the Model represents data to be displayed on the screen. More generally, Model is a Domain model that contains the business logic, data to be manipulated and data access objects.
2. **Details**:
    1. **Domain Model**: Contains the business logic and the rules that govern how the data can be manipulated.
    2. **Data Access Objects (DAOs)**: Used to interact with the database or any other storage mechanism.
    3. **Data Representation**: Holds the data that needs to be displayed on the screen.
3. **Independence**:
    1. The Model is independent of the View and the Controller. It does not need to know about their existence, promoting loose coupling.

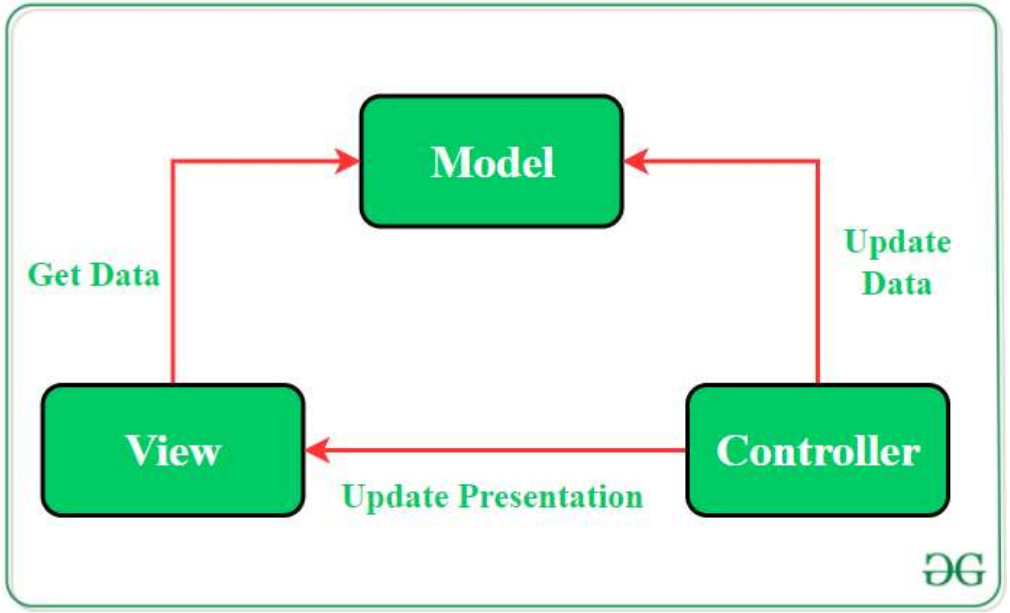


Source of the image geeksforgeeks.org

View

it also provides the visualization of the data stored in the Model and offers interaction to the user.

1. **Role**:
    1. The View is responsible for displaying the data to the user.
    2. It provides a visual representation of the Model.
2. **Details**:
    1. **UI Components**: Can include buttons, text fields, lists, etc.
    2. **Data Display**: The View retrieves data from the Model through queries and displays it to the user.
    3. **State Representation**: Shows the current state of the data and can change its state based on user interaction.

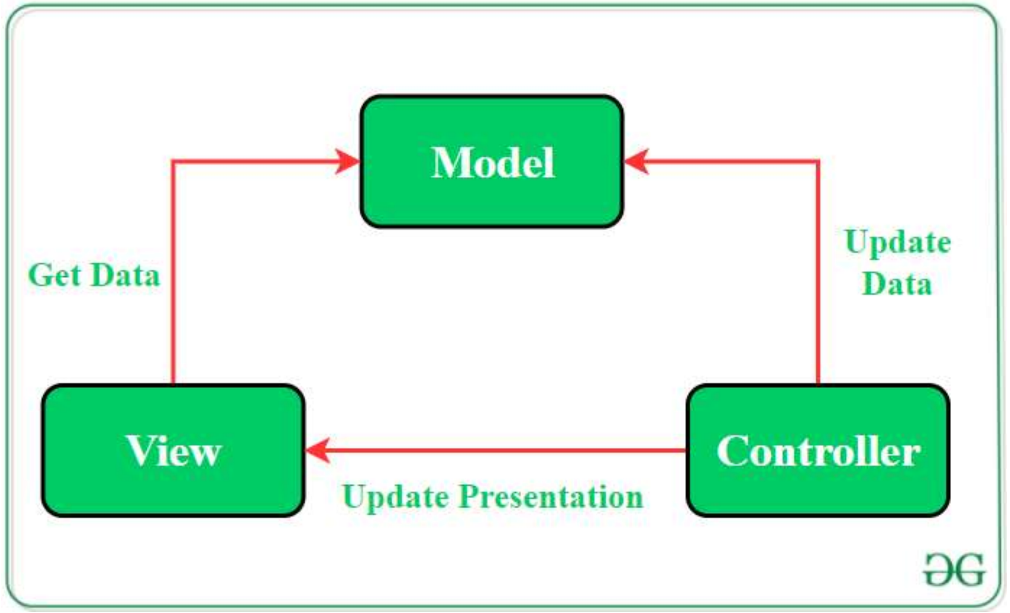


Source of the image geeksforgeeks.org

Controller

In this pattern, **View** and **Controller** both depend upon the **Model.**

1. **Role**:
   1. The Controller acts as an intermediary between the **Model** and the **View**. It handles user input and updates the Model accordingly.
   2. This component establishes the relationship between the **View** and the **Model**. It contains the core application logic and gets informed of the user’s behavior and updates the Model as per the need.
2. **Details**:
   1. **Event Handling**: Listens to user actions (e.g., clicks, form submissions) and processes them.
   2. **Model Update**: Communicates with the Model to update the state based on user actions.
   3. **View Manipulation**: Can directly update the

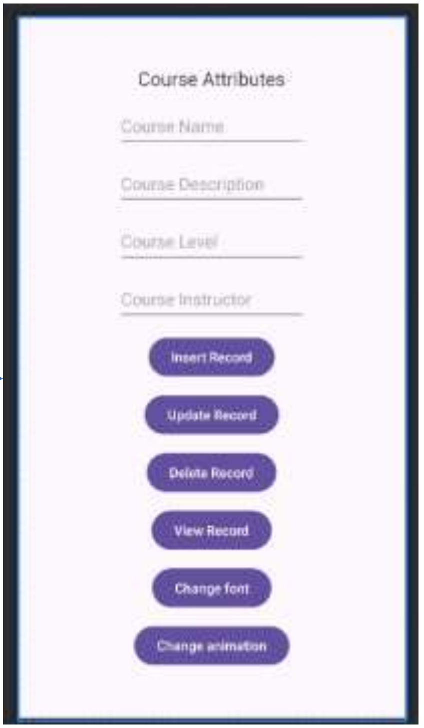


Source of the image geeksforgeeks.org

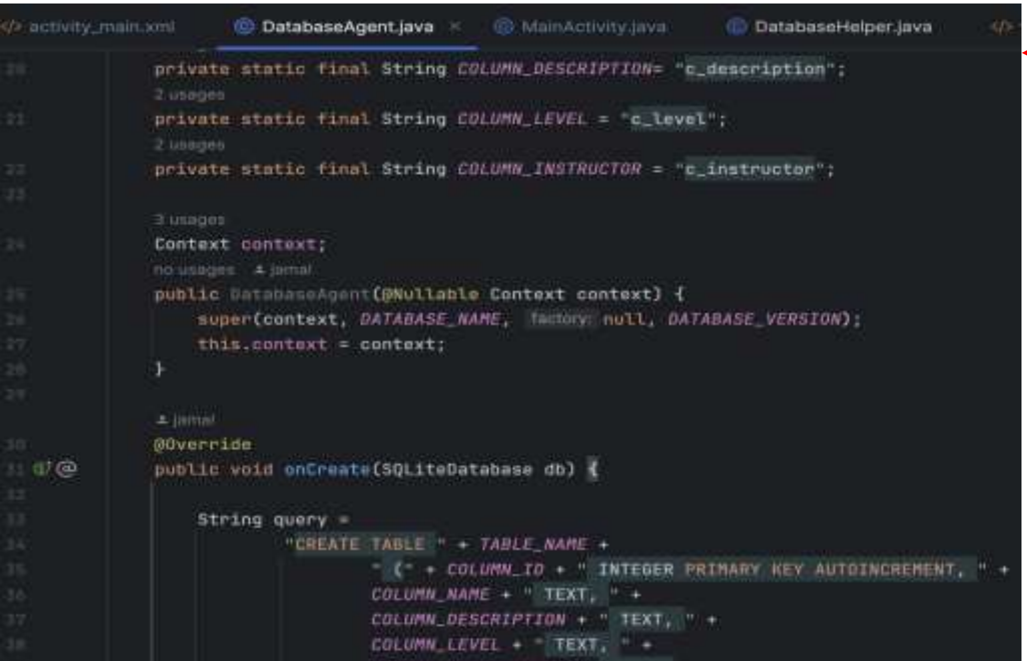
Controller

- The Controller handles events from user actions and communicates with the Model.
- The Controller also communicates with the View directly if the Model does not need to be changed (e.g., scrolling action).
- The View and the Controller depend on the Model, but the Model is completely independent.
- The design pattern states that all Views should have a single Controller, but one Controller can be shared by several Views.

View



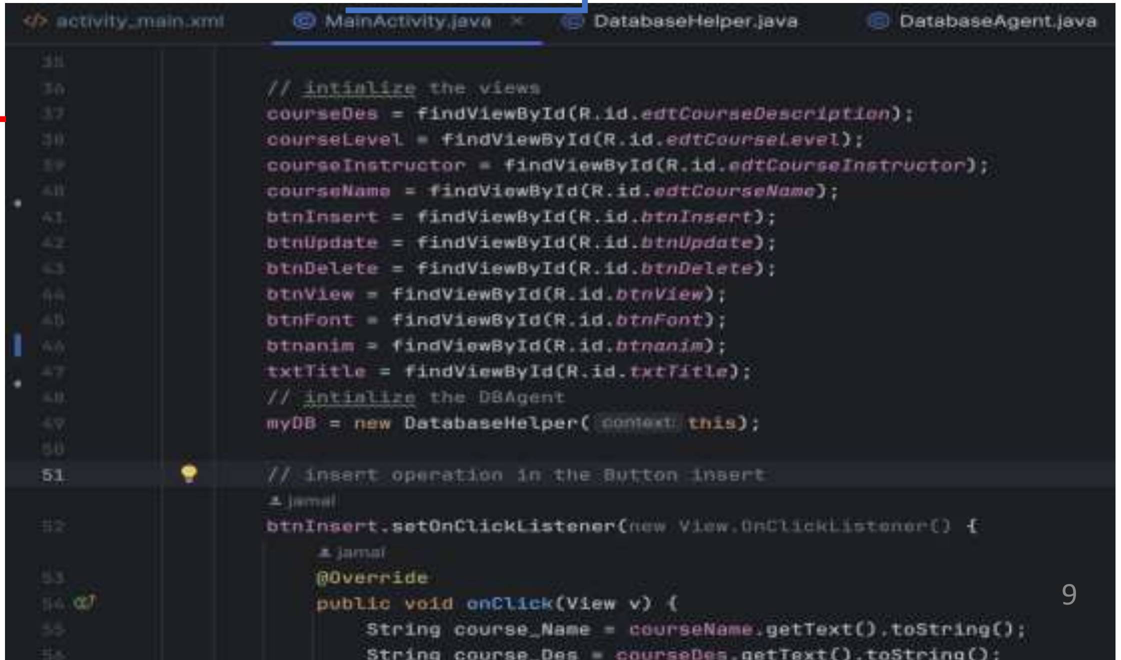
Model



```
activity_main.xml    DatabaseAgent.java ×    MainActivity.java    DatabaseHelper.java

        private static final String COLUMN_DESCRIPTION= "c_description";
        2 usages
        private static final String COLUMN_LEVEL = "c_level";
        2 usages
        private static final String COLUMN_INSTRUCTOR = "c_instructor";

        3 usages
        Context context;
        no usages  jamal
        public DatabaseAgent(@Nullable Context context) {
            super(context, DATABASE_NAME, factory: null, DATABASE_VERSION);
            this.context = context;
        }

        jamal
        @Override
        public void onCreate(SQLiteDatabase db) {

            String query =
                    "CREATE TABLE " + TABLE_NAME +
                            " (" + COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                            COLUMN_NAME + " TEXT, " +
                            COLUMN_DESCRIPTION + " TEXT, " +
                            COLUMN_LEVEL + " TEXT, " +
```

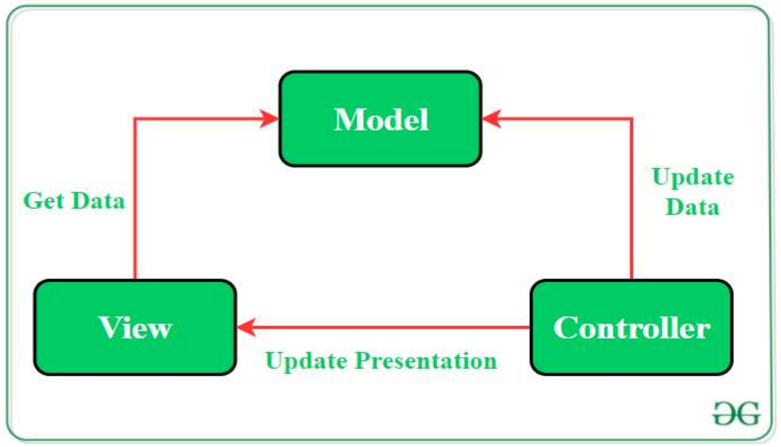
Not always MVC but as an example

Controller



```
activity_main.xml    MainActivity.java ×    DatabaseHelper.java    DatabaseAgent.java

        // intialize the views
        courseDes = findViewById(R.id.edtCourseDescription);
        courseLevel = findViewById(R.id.edtCourseLevel);
        courseInstructor = findViewById(R.id.edtCourseInstructor);
        courseName = findViewById(R.id.edtCourseName);
        btnInsert = findViewById(R.id.btnInsert);
        btnUpdate = findViewById(R.id.btnUpdate);
        btnDelete = findViewById(R.id.btnDelete);
        btnView = findViewById(R.id.btnView);
        btnFont = findViewById(R.id.btnFont);
        btnanim = findViewById(R.id.btnanim);
        txtTitle = findViewById(R.id.txtTitle);
        // intialize the DBAgent
        myDB = new DatabaseHelper( context: this);

        // insert operation in the Button insert
        jamal
        btnInsert.setOnClickListener(new View.OnClickListener() {
            jamal
            @Override
            public void onClick(View v) {
                String course_Name = courseName.getText().toString();
                String course_Des = courseDes.getText().toString();
```

Interaction Between MVC Components



1. **User Interaction with View**:
   1. The user interacts with the View by performing actions such as clicking a button or entering data into a form.
   2. Example: A user clicks a "Submit" button on a form.
2. **View Delegates to Controller**:
   1. The View sends the user actions to the Controller.
   2. Example: The click event is sent to the Controller.
3. **Controller Updates Model**:
   1. The Controller processes the input and updates the Model accordingly.
   2. Example: The Controller takes the form data and saves it to the database through the Model.
4. **Model Notifies View**:
   1. The Model notifies the View of any state changes.
   2. Example: The Model notifies that the data has been successfully saved.
5. **View Updates Itself**:
   1. The View retrieves the updated data from the Model and refreshes the UI.
   2. Example: The View updates the form to show a success message.

Advantages of MVC

- **Separation of Concerns**:
  - Each component has a distinct responsibility, making the application easier to manage and scale.
- **Modularity**:
  - Components can be developed, tested, and maintained independently.
- **Reusability**:
  - Components can be reused across different parts of the application or even in different applications.
- **Maintainability**:
  - Changes in one component have minimal impact on others, making the application easier to maintain.

•**ASP.NET MVC**: A Microsoft framework for building web applications using the MVC pattern.

Read more in here with examples: blog

Model-View-Presenter (MVP)

- **What is it:**
  - MVP is a design pattern introduced in 1996 as an adaptation of Model-View-Controller (MVC) to better suit the needs of modern, event-driven systems. MVP emphasizes the separation of concerns in an application by dividing it into three distinct components: **Model**, **View**, and **Presenter**.
- **Goals of MVP**
  - The primary goal of MVP is to **separate the business logic from the presentation logic**, similar to MVC, but with a stronger focus on the separation between the View and the Presenter.
  - This pattern ensures that the View is passive and only responsible for displaying data and routing user commands to the Presenter, which handles the presentation logic.

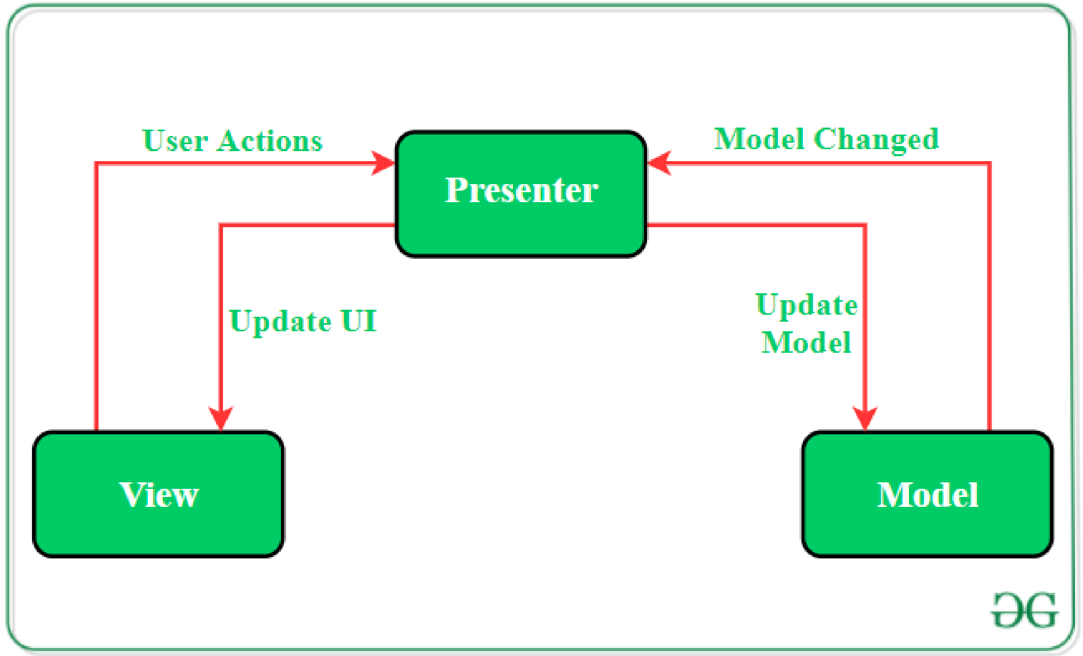


Image source: blog

Model

•**Model:** Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

1. **Role**:
   1. The Model handles the data and business logic of the application.
2. **Responsibilities**:
   1. **Data Management**: Manages the data, logic, and rules of the application.
   2. **Business Logic**: Contains the business rules and processes the data.
3. **Independence**:
   1. The Model is independent of both the View and the Presenter.

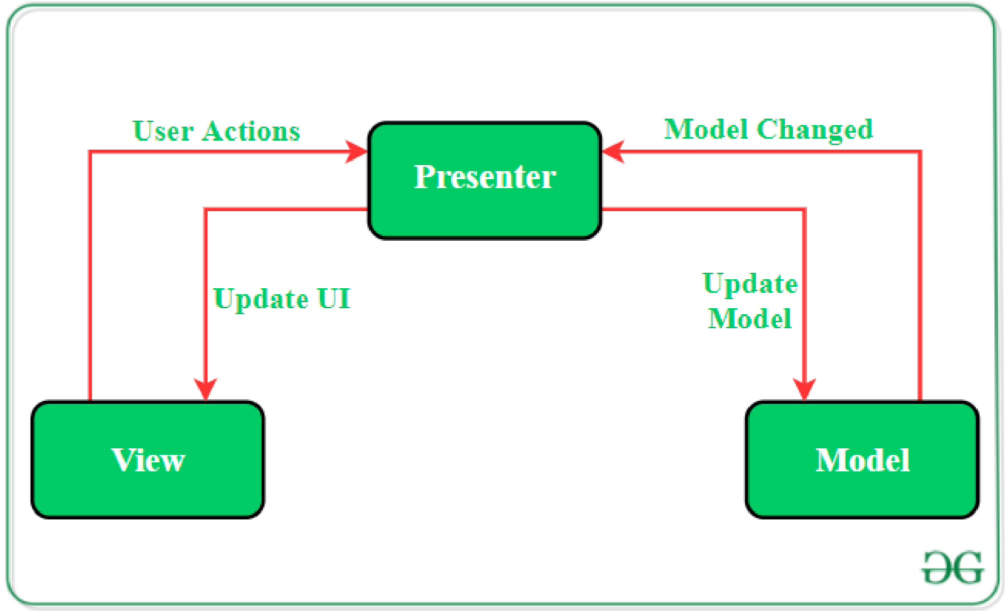


Image source: blog

View

•**View:** UI(User Interface) layer. It provides the visualization of the data and keep a track of the user's action in order to notify the Presenter.

1. **Role**:
   1. The View is a passive interface responsible for displaying data to the user and routing user commands to the Presenter.
2. **Responsibilities**:
   1. **User Interface**: Displays the data provided by the Presenter.
   2. **Event Handling**: Captures user interactions (e.g., clicks, input) and delegates them to the Presenter.
3. **Passive Nature**:
   1. The View does not contain any logic to manipulate the data;
   2. it simply passes user commands to the Presenter and updates its display based on the Presenter's instructions.

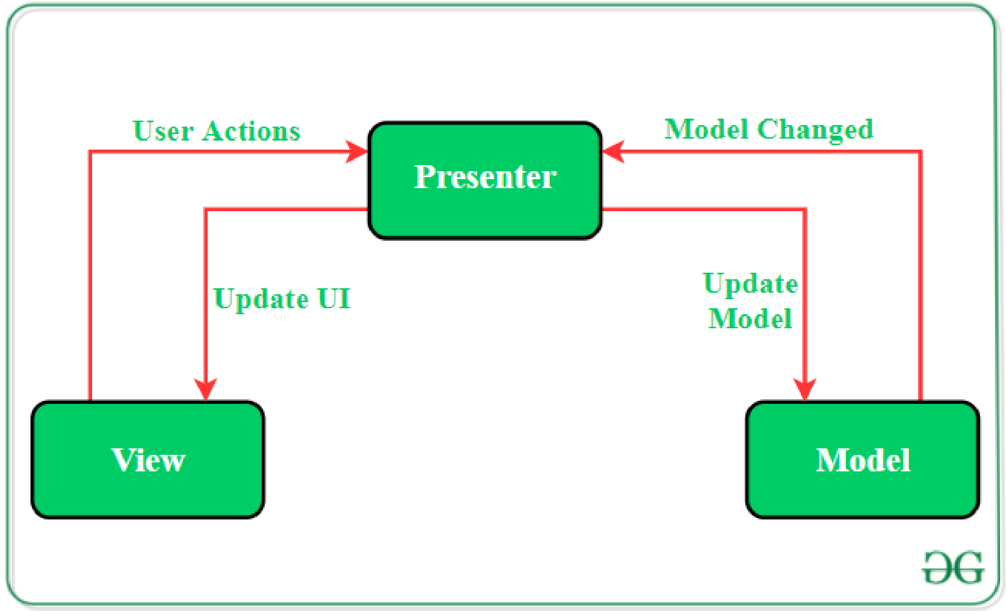
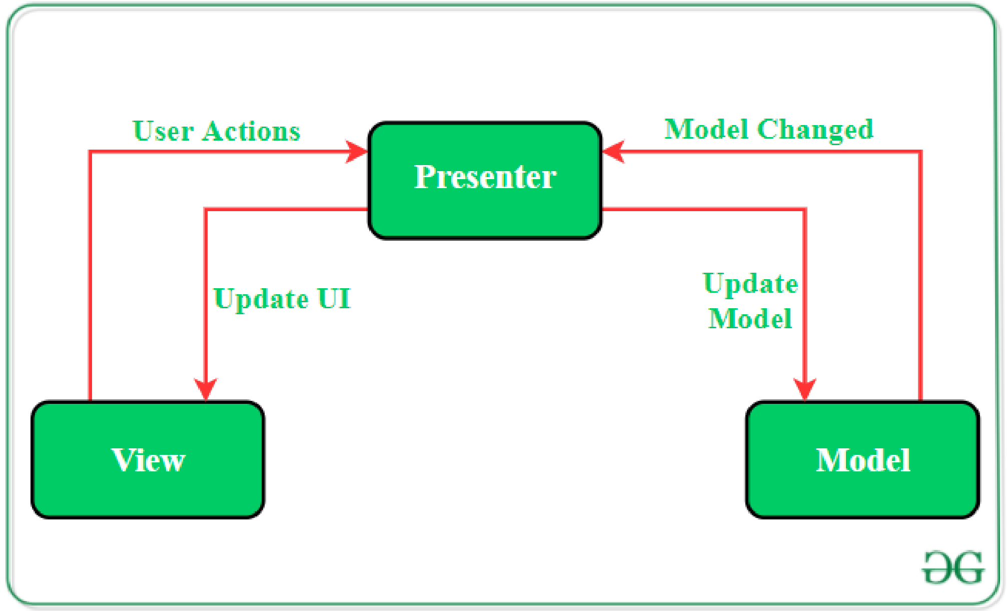


Image source: blog

Presenter

1. **Role**:
   1. The Presenter acts as an intermediary between the View and the Model.
   2. It retrieves data from the Model, processes it, and formats it for display in the View.
2. **Responsibilities**:
   1. **Presentation Logic**: Handles the logic for displaying data in the View.
   2. **Data Processing**: Processes user input, updates the Model, and retrieves updated data.
   3. **View Updates**: Updates the View with the new data after processing it.
3. **Interactions**:
   1. **With Model**: The Presenter updates and retrieves data from the Model.
   2. **With View**: The Presenter updates the View based on the data and state changes in the Model.

- **Presenter:** Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user's input notification from the View.

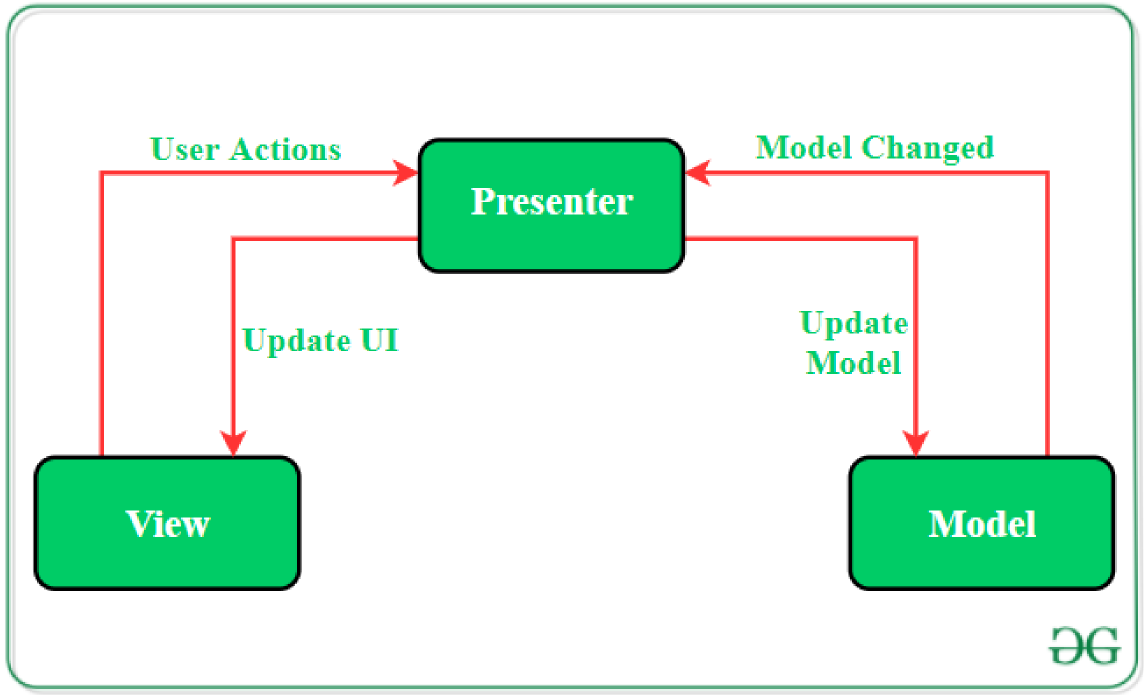


contains core business logic which will decide what to display and how to display.

Image source: blog

Key Points of MVP Architecture

1. Communication between View-Presenter and Presenter-Model happens via an **interface(also called Contract)**.
2. One Presenter class manages one View at a time i.e., there is a one-to-one relationship between Presenter and View.
3. Model and View class doesn't have knowledge about each other's existence.

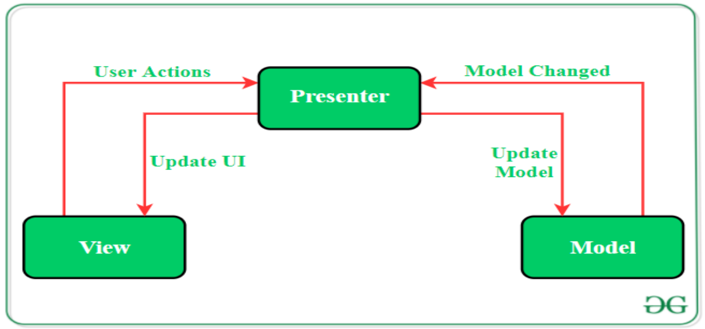


The **Presenter** must explicitly call methods on the View to update it, making the update process more manual and code-intensive.

•**MVP:** The Presenter holds a reference to the View, leading to a tighter coupling between these components.

Image source: blog

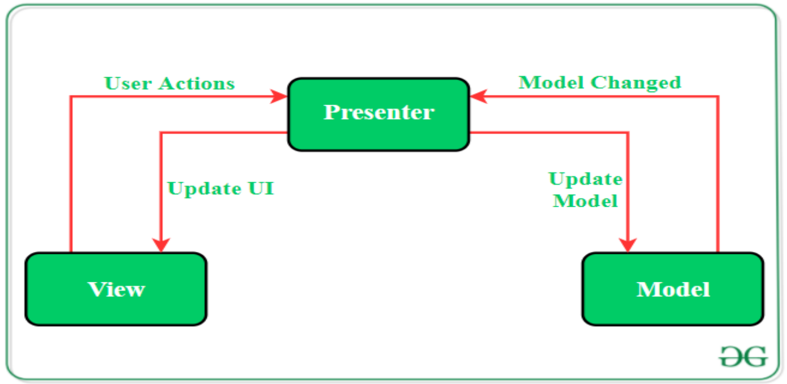
Interaction Flow in MVP



1. **User Interaction with View**:
   1. The user interacts with the View by performing actions such as clicking a button or entering data into a form.
   2. Example: A user clicks a "Submit" button on a form.
2. **View Delegates to Presenter**:
   1. The View captures the user action and delegates it to the Presenter.
   2. Example: The click event is passed to the Presenter.
3. **Presenter Processes Input**:
   1. The Presenter processes the user input, updates the Model if necessary, and retrieves the updated data.
   2. Example: The Presenter takes the form data, validates it, and saves it to the Model.
4. **Presenter Updates View**:
   1. The Presenter formats the retrieved data and updates the View accordingly.
   2. Example: The Presenter updates the View to show a success message or refreshed data.

Image source: blog

Advantages of MVP



- **Separation of Concerns**:
  - Clearly separates the presentation logic from the UI logic, making the code easier to manage and test.
- **Testability**:
  - The Presenter can be tested independently of the View and the Model.

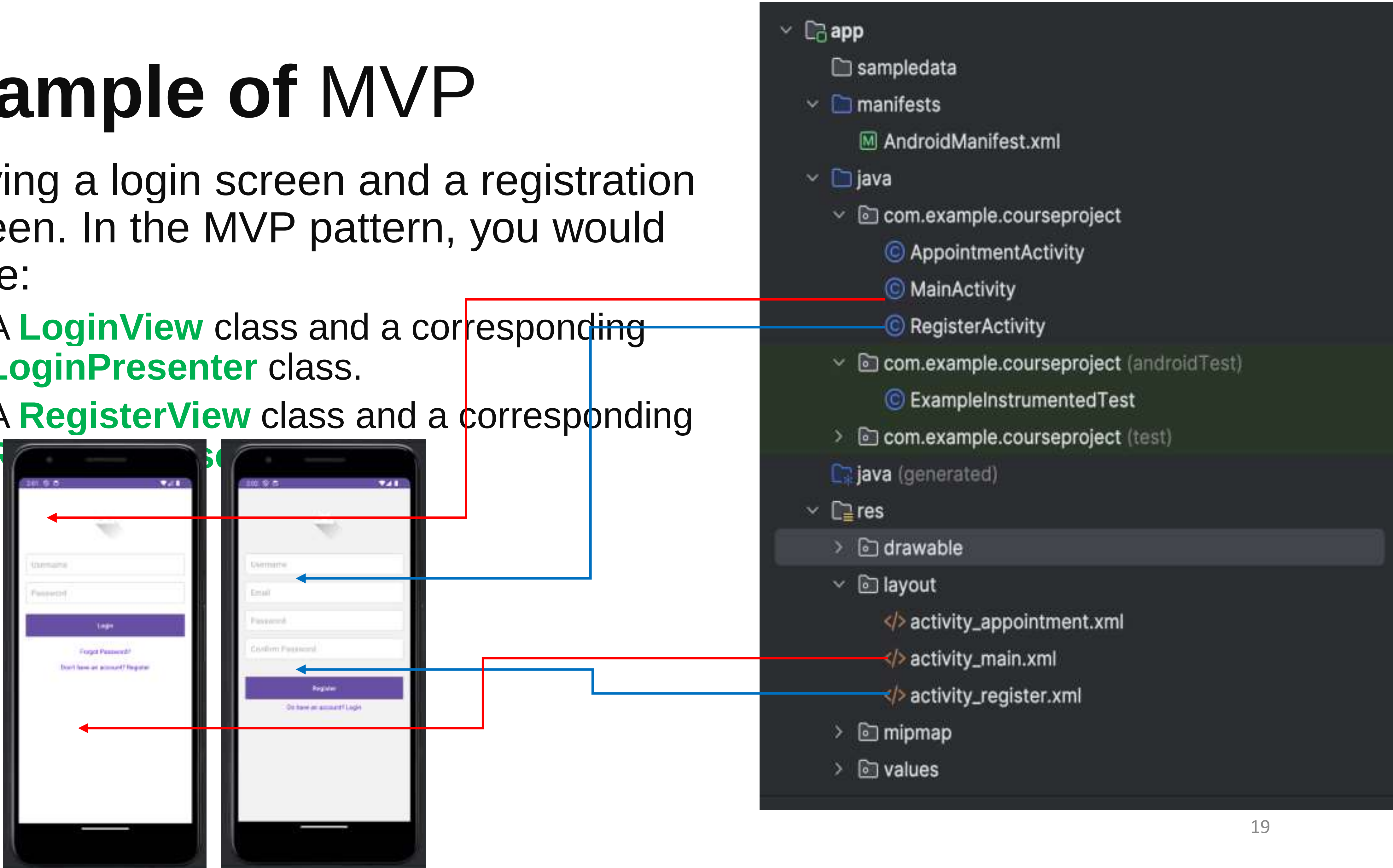
  1. **Simplified Testing:**
     1. Since a Presenter is responsible for only one View, unit tests can be more focused and straightforward, ensuring that the presentation logic for that View is correctly implemented.
- **Flexibility**:
  - The View can be easily changed without affecting the business logic.

- **Frameworks Supporting MVP**
  - **Android Development (using native Android)**:
    - MVP is widely used in Android applications to separate the UI logic from the business logic.

Get more examples in here: blog

Example of MVP

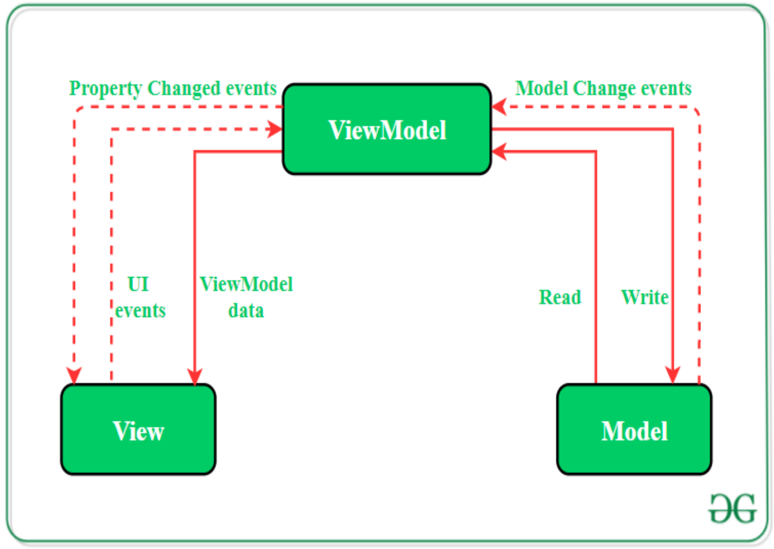
- Having a login screen and a registration screen. In the MVP pattern, you would have:
  - A **LoginView** class and a corresponding **LoginPresenter** class.
  - A **RegisterView** class and a corresponding **R**



```
app
  sampledata
  manifests
    AndroidManifest.xml
  java
    com.example.courseproject
      AppointmentActivity
      MainActivity
      RegisterActivity
    com.example.courseproject (androidTest)
      ExampleInstrumentedTest
    com.example.courseproject (test)
  java (generated)
  res
    drawable
    layout
      activity_appointment.xml
      activity_main.xml
      activity_register.xml
    mipmap
    values
```

Model-View-ViewModel (MVVM)

- **What is it**:
  - MVVM is aim is to decouple the user interface from the business logic in an application.
  - It builds on the principles of the MVP pattern, introducing a new component, the ViewModel, to act as a more refined intermediary between the Model and the View.
- **Goals of MVVM**
  - The primary goal of MVVM is to provide a clear separation between the user interface (UI) and the business logic, making it easier to manage and test each component independently. This pattern emphasizes the use of two-way data binding between the View and the ViewModel, allowing for automatic synchronization of the UI with the underlying data.

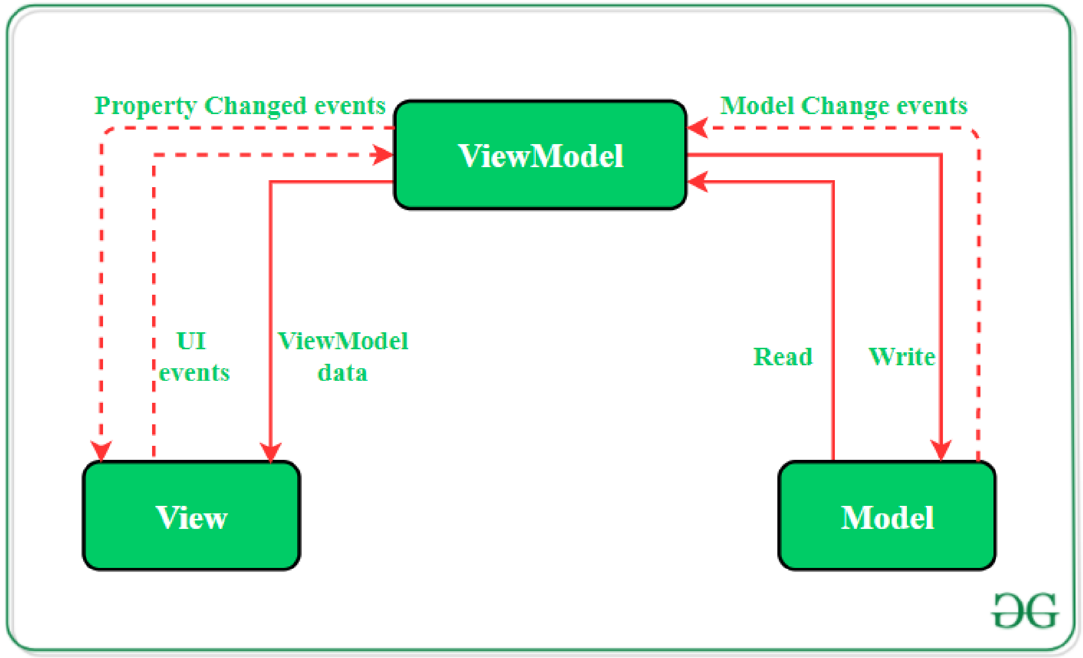


MVVM suggests separating the data presentation logic(Views or UI) from the core business logic part of the application.

Model

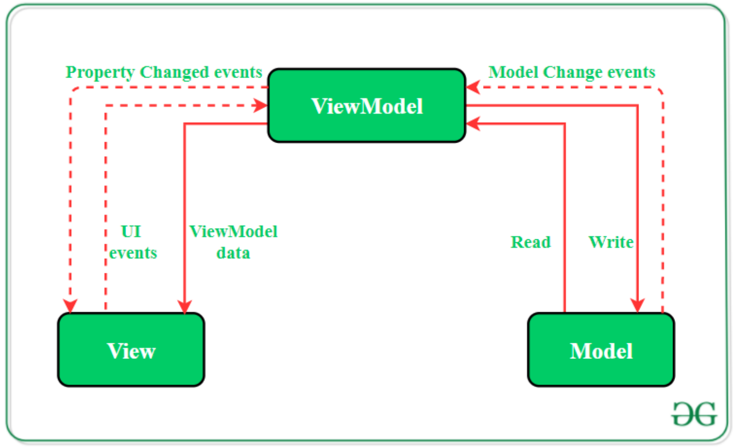
•**Model:** This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

- **Role**:
  - The Model handles the data and business logic of the application.
- **Responsibilities**:
  - **Data Management**: Manages the data, logic, and rules of the application.
  - **Business Logic**: Contains the business rules and processes the data.



View

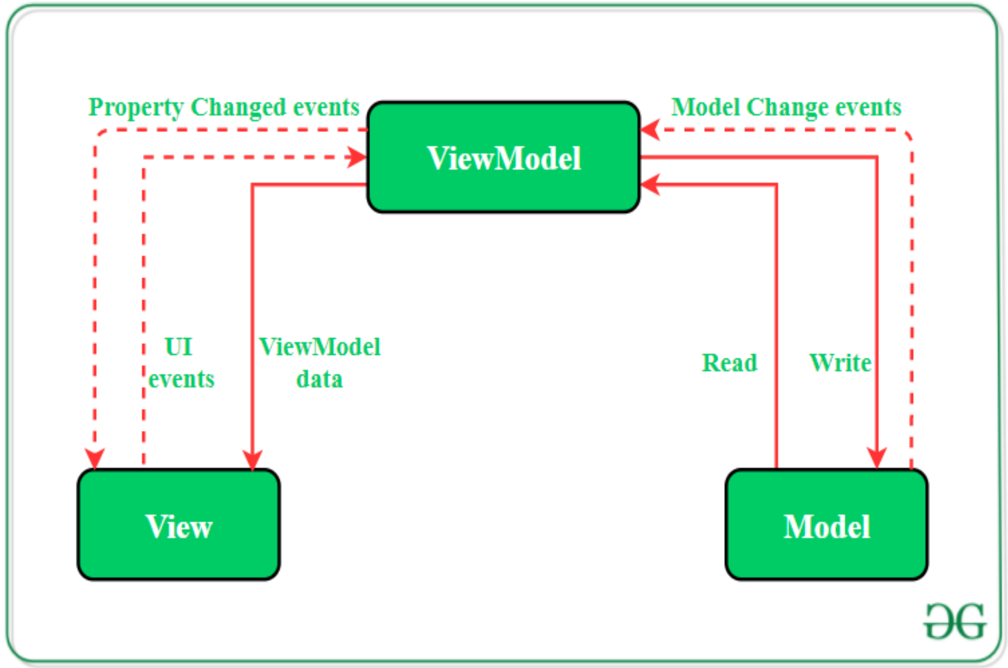
•**View:** The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.



1. **Role**:
   1. The View is responsible for displaying data to the user and notifying the ViewModel about user interactions.
2. **Responsibilities**:
   1. **User Interface**: Displays the data provided by the ViewModel.
   2. **Event Handling**: Captures user interactions (e.g., clicks, input) and communicates them to the ViewModel.
3. **Passive Nature**:
   1. The View is as passive as possible, declaring only UI elements and delegating all logic to the ViewModel.

ViewModel

•**ViewModel:** It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.



1. **Role**:
   1. The ViewModel acts as an intermediary between the View and the Model. It holds and processes data from the Model and prepares it for the View.
2. **Responsibilities**:
   1. **Data Binding**: Synchronizes the View with the data using two-way data binding.
   2. **Data Transformation**: Transforms Model data into a format suitable for the View.
   3. **Event Handling**: Implements user input events and updates the Model accordingly.
3. **Independence**:
   1. The ViewModel should not reference the View directly, allowing it to be reused with different Views without breaking the code.

Data Binding in MVVM

- **Data Binding:**
  - Data binding is a technique that connects the UI elements (View) directly to data sources (ViewModel), allowing for automatic updates to the UI when the data changes.
  - In the context of MVVM, data binding simplifies the synchronization between the View and the ViewModel by automatically updating the UI when the ViewModel properties change and vice versa.
- **Two-Way Data Binding:**
  - **Two-way data binding** means that changes in the UI automatically update the underlying data model and changes in the data model automatically update the UI.
  - This creates a seamless flow of data between the View and the ViewModel, ensuring that the UI always reflects the current state of the data and that user interactions are immediately propagated back to the data model.

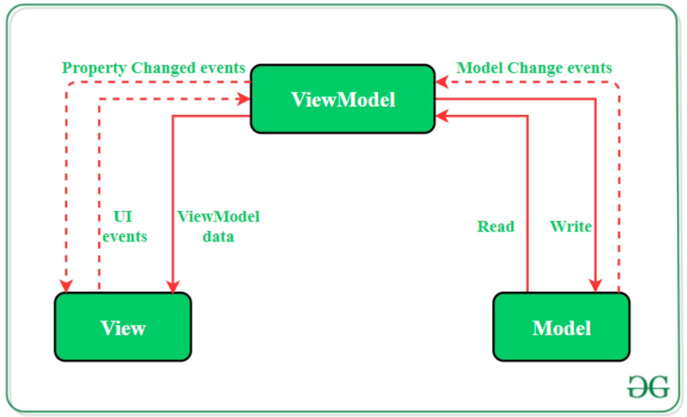
MVVM similarities with the MVP

- MVVM pattern has some similarities with the MVP(Model — View — Presenter) design pattern as the Presenter role is played by ViewModel. However, the drawbacks of the MVP pattern has been solved by MVVM in the following ways:

  1. ViewModel does not hold any kind of reference to the View.
  2. Many to-1 relationships exist between View and ViewModel.
     1. multiple views can share the same ViewModel.
  3. No triggering methods to update the View.
     1. No explicit methods are needed to update the View.
     2. The View automatically updates itself when the underlying data in the ViewModel changes, thanks to data binding.

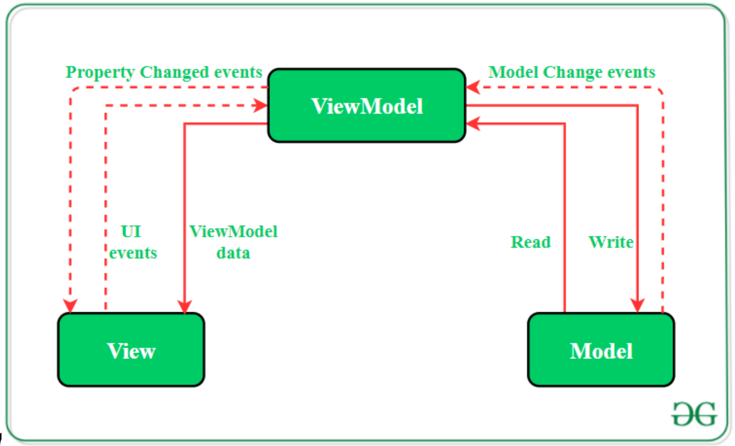
Source: blog

Interaction Flow in MVVM



1. **User Interaction with View**:
   1. The user interacts with the View by performing actions such as clicking a button or entering data into a form.
   2. Example: A user clicks a "Submit" button on a form.
2. **ViewModel Observes Interactions**:
   1. The ViewModel observes these interactions through data binding and updates the Model as needed.
   2. Example: The ViewModel takes the form data and updates the Model.
3. **Model Notifies ViewModel of Changes**:
   1. The Model notifies the ViewModel of any changes in the data.
   2. Example: The Model sends a notification that the data has been updated.
4. **ViewModel Updates View**:
   1. The ViewModel updates the View using data binding, ensuring the UI reflects the latest data.
   2. Example: The ViewModel updates the View to show a success message or refreshed data.

Benefits of MVVM



- **Separation of Concerns**:
  - Clear separation between UI and business logic.
- **Two-Way Data Binding**:
  - Automatically synchronizes the View with the ViewModel, reducing boilerplate code.
- **Testability**:
  - The ViewModel can be tested independently of the View.
- **Reusability**:
  - The ViewModel can be reused with different Views.

Comparison with MVC MVP, and MVVM

- **View's Role**:
  - **MVC**: The View handles user input and updates the Model directly.
  - **MVP**: The View is a passive interface and delegates all user interactions to the Presenter.
  - **MVVM**: The View is passive and uses data binding to synchronize with the ViewModel.
- **Controller/Presenter vs. ViewModel**:
  - **MVC**: The Controller handles user input, updates the Model, and updates the View.
  - **MVP**: The Presenter acts as an intermediary, processing input, updating the Model, and updating the View.
  - **MVVM**: The ViewModel acts as an intermediary, holding and processing data, and synchronizing the View **through data binding.**
- **Interaction Flow**:
  - **MVC**: User interacts with the View -> View updates Model -> Model updates View.
  - **MVP**: User interacts with the View -> View delegates to Presenter -> Presenter updates Model -> Presenter updates View.
  - **MVVM**: User interacts with the View -> ViewModel observes and updates Model -> Model notifies ViewModel -> ViewModel updates View through data binding.

Differences Between MVC, MVP, and MVVM

- **Separation of Concerns**:
  - **MVC**: The Controller handles user input and updates the View and Model. The View is directly updated by the Model.
  - **MVP**: The Presenter handles user input, updates the Model, and updates the View.
  - **MVVM**: The ViewModel handles user input, updates the Model, and updates the View via data binding.
- **View Updates**:
  - **MVC**: The View listens to changes in the Model.
  - **MVP**: The Presenter updates the View directly.
  - **MVVM**: The ViewModel uses data binding to automatically update the View.
- **Complexity**:
  - **MVC**: Simpler but can become complex with large applications.
  - **MVP**: More complex than MVC, provides better separation.
  - **MVVM**: Most complex due to data binding, but provides the best separation and is highly testable.
- **Framework Support**:
  - **MVC**: AngularJS (early versions), Ruby on Rails, ASP.NET MVC.
  - **MVP**: Android native development, GWT.
  - **MVVM**: Angular, React with Redux, Vue.js, WPF.