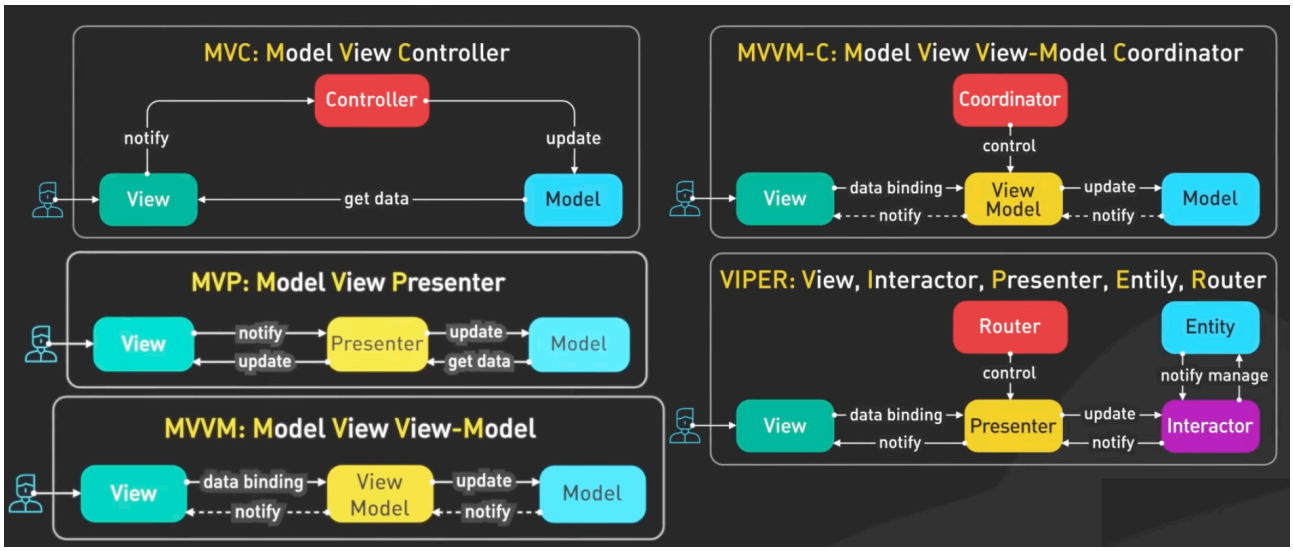
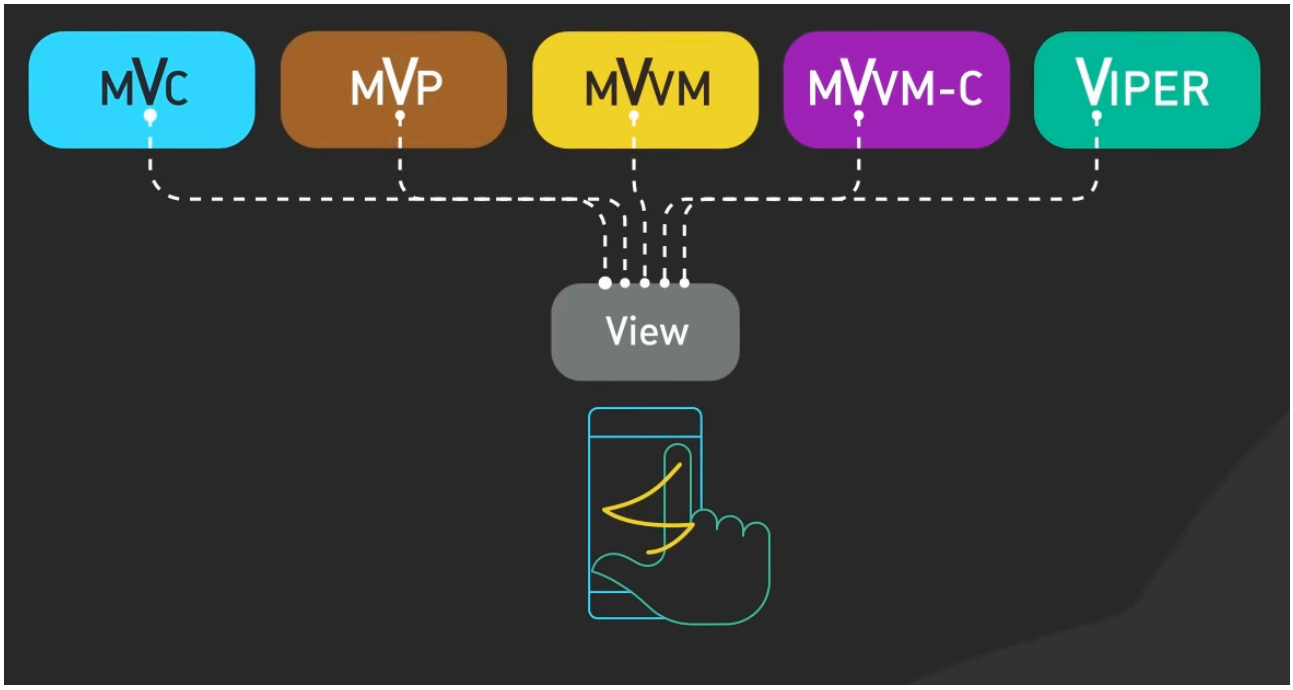
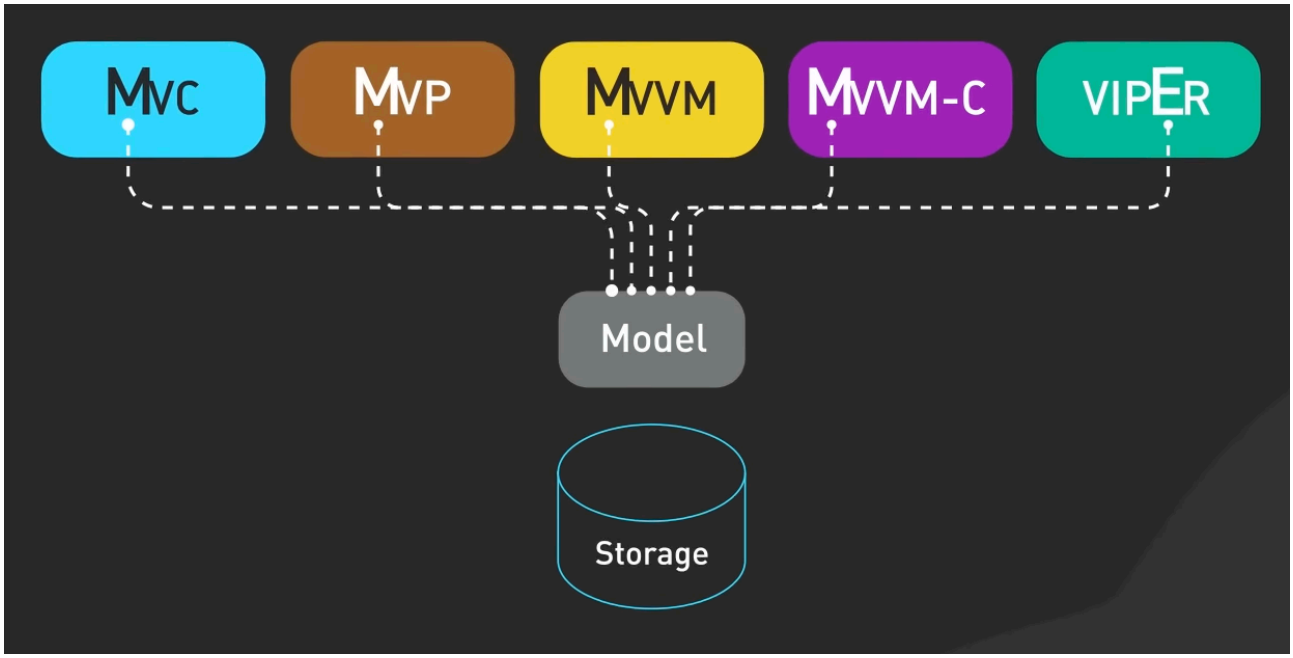
App Architectural Patterns







⭐ MVC — Model–View–Controller (Full Explanation)

MVC is a **software architectural pattern** that separates an application into **three layers**:

1. Model (M)

The **Model** holds your **data**, **business logic**, and **rules**.

It is responsible for:

- Fetching data (DB/API)
- Updating data
- Enforcing business rules
- Notifying the controller when data changes (in classic MVC)

Examples:

- A `User` struct/class
- A database table representation
- A form validator
- A network manager

Model should not know about UI.

2. View (V)

The **View** handles **UI**, such as screens, HTML pages, Android XML layouts, SwiftUI views, or templates.

It is responsible for:

- Displaying information
- Receiving user input events (taps, clicks)
- Letting the controller know when something happens

View should not contain business logic.

3. Controller (C)

The **Controller** is the **middleman**.

It is responsible for:

- Receiving user input from the view
- Updating the model
- Getting new data from the model
- Updating the view

Controller coordinates everything, but should not store UI code or business rules.

📌 Why MVC?

✔ Clean separation of responsibilities
✔ Testable
✔ Easier to scale
✔ Makes teamwork easier
✔ Reduces coupling

⭐ EXAMPLE 1 — Swift (UIKit) MVC

Model.swift

```
import Foundation

struct User {
    let id: Int
    var name: String
}

class UserRepository {
    private var users = [
        User(id: 1, name: "Nelson"),
        User(id: 2, name: "Cathy")
    ]

    func getUsers() -> [User] {
        return users
```

```
    }
}
```

View: Main.storyboard

A table view.

Controller.swift

```
import UIKit

class UserController: UIViewController, UITableViewDataSource
{

    @IBOutlet weak var tableView: UITableView!

    let repo = UserRepository()
    var users: [User] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.dataSource = self

        // Controller asks Model for data
        users = repo.getUsers()
        tableView.reloadData()
    }

    // MARK: - Table View
    func tableView(_ tableView: UITableView,
                   numberOfRowsInSection section: Int) -> Int
{
        return users.count
    }

    func tableView(_ tableView: UITableView,
                   cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        let cell =
tableView.dequeueReusableCell(withIdentifier: "cell",
                                                  for:
indexPath)
        cell.textLabel?.text = users[indexPath.row].name
```

```
        return cell
    }
}
```

Flow:

1. Controller loads
2. Asks Model for users
3. Updates View

⭐ EXAMPLE 2 — Kotlin Android (MVC-style)

Android isn't naturally MVC, but this is the closest representation.

Model.kt

```
data class User(val id: Int, val name: String)

class UserRepository {
    fun getUsers(): List<User> {
        return listOf(
            User(1, "Nelson"),
            User(2, "Cathy")
        )
    }
}
```

View → activity_main.xml

```
<ListView
    android:id="@+id/userList"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Controller → MainActivity.kt

```
class MainActivity : AppCompatActivity() {
```

```
    private val repo = UserRepository()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val listView = findViewById<ListView>(R.id.userList)

        val users = repo.getUsers()
        val names = users.map { it.name }

        val adapter = ArrayAdapter(
            this,
            android.R.layout.simple_list_item_1,
            names
        )

        listView.adapter = adapter
    }
}
```

⭐ EXAMPLE 3 — React Frontend + Express MVC Backend

React is not MVC, but your **backend** can be.

Folder structure:

```
/controllers
    userController.js
/models
    userModel.js
/routes
    userRoutes.js
app.js
```

models/userModel.js

```
const users = [
  { id: 1, name: "Nelson" },
  { id: 2, name: "Cathy" }
```

```
];

exports.getAllUsers = () => {
  return users;
};
```

controllers/userController.js

```
const UserModel = require("../models/userModel");

exports.getUsers = (req, res) => {
  const data = UserModel.getAllUsers();
  res.json(data);
};
```

routes/userRoutes.js

```
const express = require("express");
const router = express.Router();
const userController = require("../controllers/
userController");

router.get("/", userController.getUsers);

module.exports = router;
```

app.js

```
const express = require("express");
const userRoutes = require("./routes/userRoutes");

const app = express();
app.use("/users", userRoutes);

app.listen(3000, () => console.log("Server running on port
3000"));
```

⭐ EXAMPLE 4 — Python Flask (MVC-Style)

Flask is **not strictly MVC**, but you can structure it as such.

Folder structure:

```
/models/user.py
/controllers/user_controller.py
/templates/users.html
app.py
```

models/user.py

```
class User:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class UserRepository:
    def get_users(self):
        return [
            User(1, "Nelson"),
            User(2, "Cathy"),
        ]
```

controllers/user_controller.py

```
from flask import Blueprint, render_template
from models.user import UserRepository

user_bp = Blueprint('user', __name__)
repo = UserRepository()

@user_bp.route("/users")
def list_users():
    users = repo.get_users()
    return render_template("users.html", users=users)
```

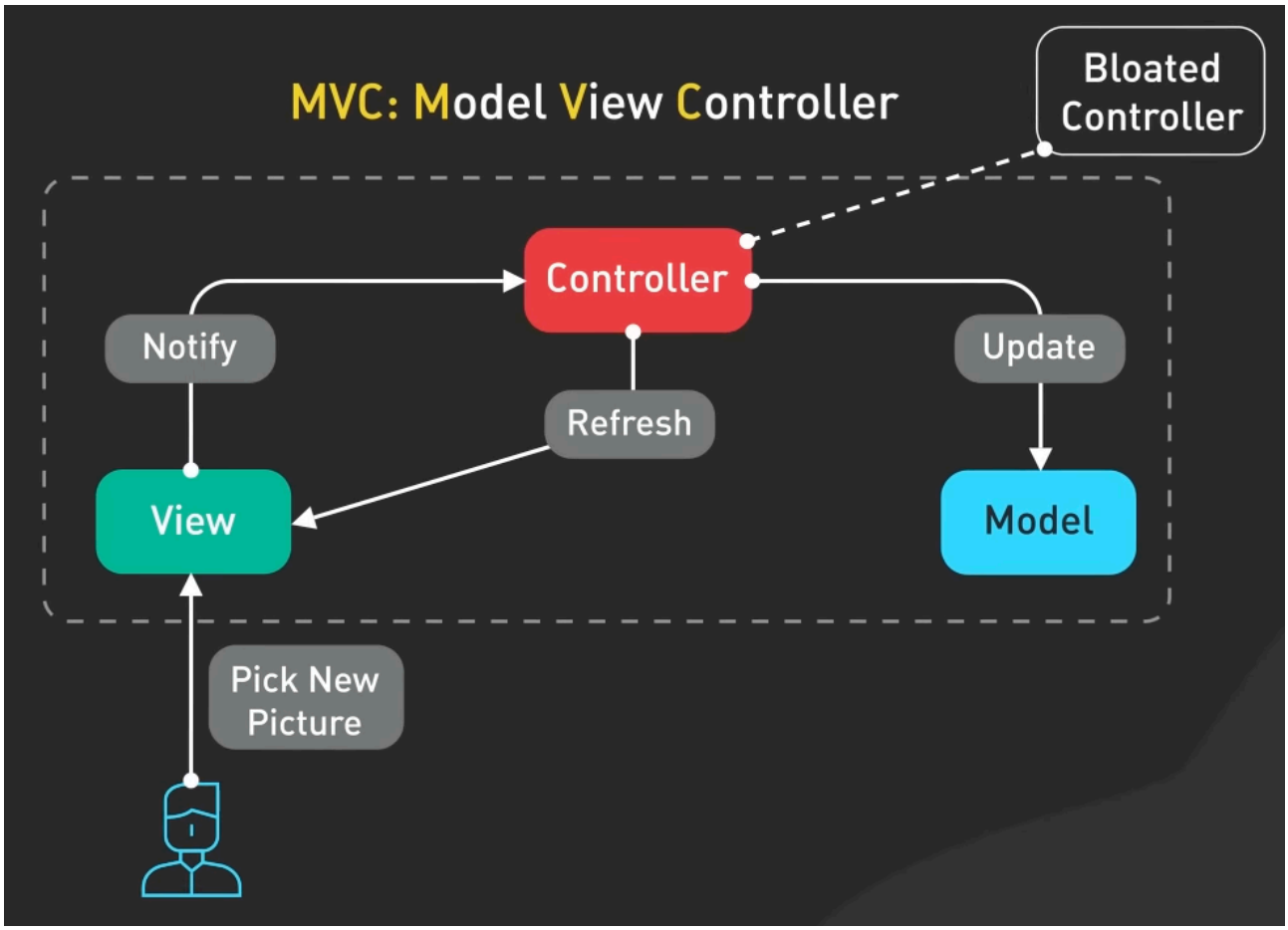
templates/users.html

```
<html>
<body>
    <h1>User List</h1>
    <ul>
```

```
        {% for user in users %}
            <li>{{ user.name }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

app.py

```
from flask import Flask
from controllers.user_controller import user_bp

app = Flask(__name__)
app.register_blueprint(user_bp)

if __name__ == "__main__":
    app.run(debug=True)
```



What is MVP?

MVP = Model–View–Presenter

It splits your app into:

- **Model** – Data + business logic (same idea as MVC)
- **View** – UI only; shows data, forwards user events
- **Presenter** – Middleman; contains almost all the "smart" logic

Key idea:

The **View is dumb**.
The **Presenter is smart**.
The **Model is data + rules**.

Roles

Model

- Holds data and business logic
- Talks to APIs, database, etc.
- **Should not** know about View or Android UI, etc.

View

- Renders the UI
- Forwards user actions to Presenter (e.g., button click → `presenter.onLoginClicked(...)`)
- Implements an interface (e.g. `LoginContract.View`) that the Presenter uses:
  - `showLoading()`
  - `showError(message)`
  - `navigateToHome()`
- Contains **no business decision-making** if possible.

Presenter

- Has a reference to **View (interface)** and **Model/Repository**
- Handles user actions:

    - Validate input
    - Call Model
    - Decide what View should display
- Does **not** contain UI classes (like Android `Context`, `Button`, etc.)—only talks through the View interface.

3. Android/Kotlin MVP Example – Login Screen

We'll build a simple **Login feature** using MVP.

Folder Structure (conceptual)

```
login/
   LoginContract.kt
   LoginPresenter.kt
   LoginActivity.kt      // View
   AuthRepository.kt     // Model
```

3.1 Contract – Define View & Presenter interfaces

```
// login/LoginContract.kt
package com.example.app.login

interface LoginContract {

    interface View {
        fun showLoading()
        fun hideLoading()
        fun showError(message: String)
        fun navigateToHome()
        fun getEmailInput(): String
        fun getPasswordInput(): String
    }

    interface Presenter {
        fun onLoginButtonClicked()
        fun onDestroy()   // to avoid memory leaks
    }
}
```

- `View` interface: everything the Presenter can ask the View to do.
- `Presenter` interface: everything the View can ask the Presenter to do.

3.2 Model – Authentication Repository

```
// login/AuthRepository.kt
package com.example.app.login

class AuthRepository {

    // Simulate login with a callback
    fun login(email: String, password: String, callback:
(Boolean, String?) -> Unit) {
        // In real life: call API, check DB, etc.
        // Here we fake an async call:
        Thread {
            Thread.sleep(1000) // simulate network delay

            if (email == "test@example.com" && password ==
"123456") {
                callback(true, null)
            } else {
                callback(false, "Invalid email or password")
            }
        }.start()
    }
}
```

- Represents your **Model** (data + logic).
- Could be replaced later with real networking.

3.3 Presenter – Core Logic

```
// login/LoginPresenter.kt
package com.example.app.login

class LoginPresenter(
    private var view: LoginContract.View?,
    private val authRepository: AuthRepository
) : LoginContract.Presenter {

    override fun onLoginButtonClicked() {
        val email = view?.getEmailInput()?.trim().orEmpty()
        val password =
view?.getPasswordInput()?.trim().orEmpty()
```

```
        // Basic validation (Presenter contains this logic)
        if (email.isEmpty()) {
            view?.showError("Email cannot be empty")
            return
        }
        if (password.isEmpty()) {
            view?.showError("Password cannot be empty")
            return
        }

        view?.showLoading()

        authRepository.login(email, password) { success,
errorMessage ->
            // We're in a background thread; in Android you'd
post to main thread
            // For simplicity, assume we're back on main
thread

            view?.hideLoading()

            if (success) {
                view?.navigateToHome()
            } else {
                view?.showError(errorMessage ?: "Unknown
error")
            }
        }
    }

    override fun onDestroy() {
        // Avoid leaking the Activity
        view = null
    }
}
```

Key points:

- Presenter gets input indirectly via `view.getEmailInput()`.
- Presenter decides validation rules and error messages.
- View only displays what Presenter tells it.

3.4 View – Activity implementing the View interface

XML Layout (View)

```
<!-- res/layout/activity_login.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    android:orientation="vertical"
    android:padding="16dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/editEmail"
        android:hint="Email"
        android:inputType="textEmailAddress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <EditText
        android:id="@+id/editPassword"
        android:hint="Password"
        android:inputType="textPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/btnLogin"
        android:text="Login"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <ProgressBar
        android:id="@+id/progressBar"
        android:visibility="gone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"/>

</LinearLayout>
```

Activity as MVP View

```
// login/LoginActivity.kt
package com.example.app.login
```

```
import android.os.Bundle
import android.view.View
import android.widget.*
import androidx.appcompat.app.AppCompatActivity

class LoginActivity : AppCompatActivity(), LoginContract.View
{

    private lateinit var presenter: LoginContract.Presenter

    private lateinit var editEmail: EditText
    private lateinit var editPassword: EditText
    private lateinit var btnLogin: Button
    private lateinit var progressBar: ProgressBar

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)

        // Find views
        editEmail = findViewById(R.id.editEmail)
        editPassword = findViewById(R.id.editPassword)
        btnLogin = findViewById(R.id.btnLogin)
        progressBar = findViewById(R.id.progressBar)

        // Create Presenter
        presenter = LoginPresenter(this, AuthRepository())

        btnLogin.setOnClickListener {
            // Delegate action to Presenter
            presenter.onLoginButtonClicked()
        }
    }

    // ---- View Interface Implementation ----

    override fun showLoading() {
        progressBar.visibility = View.VISIBLE
        btnLogin.isEnabled = false
    }

    override fun hideLoading() {
        progressBar.visibility = View.GONE
        btnLogin.isEnabled = true
```

```
    }

    override fun showError(message: String) {
        Toast.makeText(this, message,
Toast.LENGTH_SHORT).show()
    }

    override fun navigateToHome() {
        Toast.makeText(this, "Login successful! Go to Home.",
Toast.LENGTH_SHORT).show()
        // startActivity(Intent(this,
HomeActivity::class.java))
    }

    override fun getEmailInput(): String =
editEmail.text.toString()

    override fun getPasswordInput(): String =
editPassword.text.toString()

    override fun onDestroy() {
        super.onDestroy()
        presenter.onDestroy()
    }
}
```

Data Flow (Login)

1. User taps **Login** button → View calls `presenter.onLoginButtonClicked()`.
2. Presenter reads `view.getEmailInput()` / `view.getPasswordInput()`.
3. Presenter validates input.
4. Presenter calls `authRepository.login(...)`.
5. Model returns success/failure.
6. Presenter decides:
    - On success → `view.navigateToHome()`
    - On error → `view.showError(message)`
7. View simply shows UI, no business logic.

That's **classic MVP**.

4. Simple Web JavaScript MVP Example

To show the same pattern in a web context.

HTML (View)

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>MVP Counter</title>
</head>
<body>
  <h1 id="count">0</h1>
  <button id="increment">Increment</button>
  <script src="app.js"></script>
</body>
</html>
```

JavaScript – Model, View, Presenter

```
// app.js

// Model
class CounterModel {
  constructor() {
    this.value = 0;
  }

  increment() {
    this.value++;
  }

  getValue() {
    return this.value;
  }
}

// View
class CounterView {
  constructor() {
    this.countElement = document.getElementById("count");
```

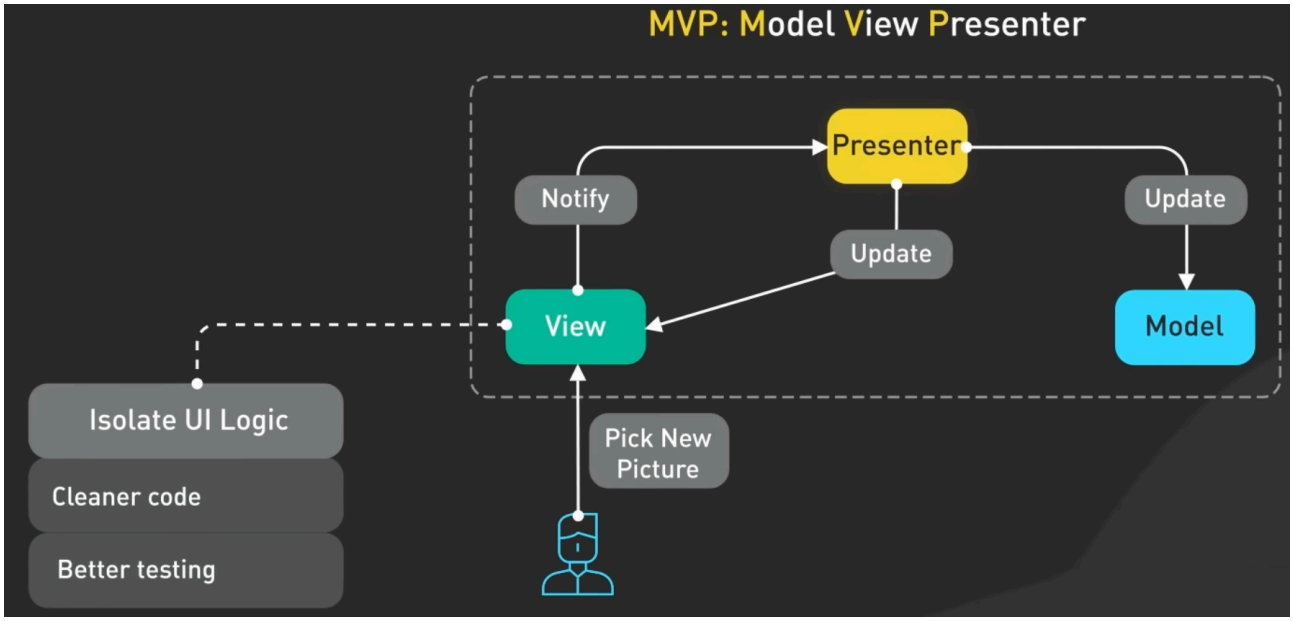
```
    this.incrementButton =
document.getElementById("increment");
  }

  bindIncrement(handler) {
    this.incrementButton.addEventListener("click", () => {
      handler();
    });
  }

  render(value) {
    this.countElement.textContent = value.toString();
  }
}

// Presenter
class CounterPresenter {
  constructor(view, model) {
    this.view = view;
    this.model = model;

    this.view.bindIncrement(this.handleIncrement.bind(this));
    this.updateView();
  }

  handleIncrement() {
    this.model.increment();
    this.updateView();
  }

  updateView() {
    const value = this.model.getValue();
    this.view.render(value);
  }
}

// Bootstrap
const model = new CounterModel();
const view = new CounterView();
const presenter = new CounterPresenter(view, model);
```

- **View** only knows DOM and `bindIncrement` + `render`.
- **Presenter** handles logic: when button clicked → update model → update view.
- **Model** only stores and updates data.

5. When to Use MVP

MVP is especially useful when:

- Your UI framework makes it hard to test Views directly.
- You want **very testable** presenters:
    - You can pass in a fake/mock `View` and `Model` in unit tests.
- You want to avoid “God Activities” / “God ViewControllers”.



🔵 MVVM — Model–View–ViewModel (Full Explanation)

MVVM is an architectural pattern designed to:

✔ Make UI predictable
✔ Decouple UI from business logic
✔ Improve testability
✔ Reduce boilerplate compared to MVC/MVP

⭐ 1. What MVVM Really Means

MVVM splits your app into three layers:

A. Model (Data + Business Rules)

- Contains business logic
- Talks to database, API, repositories
- No UI code
- Pure logic, easy to test

B. View (UI)

- Displays data
- Sends user actions (tap, click, gesture)
- Observes ViewModel and updates automatically (SwiftUI, LiveData, Vue, etc.)
- No business logic

In MVVM, the View does *not* manipulate data directly.
It only reacts to what the ViewModel provides.

C. ViewModel (The Brain)

- Connects View ↔ Model
- Exposes "UI-ready data" (e.g., formatted names, state flags)

- Holds observable data:
  - SwiftUI → `@Published`
  - Android → `LiveData` / `StateFlow`
  - Vue → reactive properties
- Contains state, logic, and transformations
- No reference to UI components (Activity/UIView etc.)

The View observes the ViewModel.
The ViewModel updates itself, and the View auto-refreshes.

⭐ Why MVVM?

✔ Cleaner than MVC

Controller becomes huge ("Massive ViewController").
MVVM moves the logic into the ViewModel.

✔ Cleaner than MVP

Presenter calls `view.showError()` → still UI-aware.
ViewModel knows nothing about UI.
View simply reacts.

✔ Perfect for modern UI frameworks

- SwiftUI (full MVVM)
- Android Jetpack ViewModel
- Vue.js (MVVM by design)

🔵 2. SwiftUI MVVM Example (Full Login Screen)

This is the cleanest and most accurate representation of MVVM.

Model — User + AuthRepository

User.swift

```
struct User {
    let id: Int
    let email: String
}
```

AuthRepository.swift

```
import Foundation

class AuthRepository {

    func login(email: String, password: String, completion:
@escaping (Result<User, Error>) -> Void) {

        DispatchQueue.global().asyncAfter(deadline: .now() +
1.0) {
            if email == "test@example.com" && password ==
"123456" {
                completion(.success(User(id: 1, email:
email)))
            } else {
                completion(.failure(NSError(domain: "", code:
401, userInfo: [
                    NSLocalizedDescriptionKey: "Invalid email
or password"
                ])))
            }
        }
    }
}
```

ViewModel — Business Logic + Observable State

LoginViewModel.swift

```
import Foundation
import Combine

class LoginViewModel: ObservableObject {

    @Published var email: String = ""
    @Published var password: String = ""

    @Published var isLoading: Bool = false
```

```
    @Published var errorMessage: String?
    @Published var isLoggedIn: Bool = false

    private let repository = AuthRepository()

    func login() {
        guard !email.isEmpty else {
            errorMessage = "Email cannot be empty"
            return
        }

        guard !password.isEmpty else {
            errorMessage = "Password cannot be empty"
            return
        }

        isLoading = true

        repository.login(email: email, password: password)
{ [weak self] result in
            DispatchQueue.main.async {
                self?.isLoading = false

                switch result {
                case .success(_):
                    self?.isLoggedIn = true
                case .failure(let error):
                    self?.errorMessage =
error.localizedDescription
                }
            }
        }
    }
}
```

Key MVVM behaviours:

- `@Published` makes View auto-update
- ViewModel contains no SwiftUI code
- ViewModel exposes UI-ready states

View — Observes ViewModel

LoginView.swift

```
import SwiftUI

struct LoginView: View {

    @StateObject private var vm = LoginViewModel()

    var body: some View {
        VStack(spacing: 16) {

            TextField("Email", text: $vm.email)
                .textFieldStyle(.roundedBorder)
                .autocapitalization(.none)

            SecureField("Password", text: $vm.password)
                .textFieldStyle(.roundedBorder)

            if vm.isLoading {
                ProgressView()
            }

            Button("Login") {
                vm.login()
            }
            .disabled(vm.isLoading)

            if let error = vm.errorMessage {
                Text(error)
                    .foregroundColor(.red)
            }

        }
        .padding()
        .alert("Login Success", isPresented: $vm.isLoggedIn)
{
            Button("OK") { }
        }
    }
}
```

This is pure MVVM:

- UI is fully reactive
- ViewModel drives View state
- Zero business logic in the View

🔵 3. Android MVVM (Kotlin + ViewModel + LiveData)

Model — AuthRepository

```
class AuthRepository {

    fun login(email: String, password: String, callback:
(Boolean, String?) -> Unit) {
        Thread {
            Thread.sleep(1000)

            if (email == "test@example.com" && password ==
"123456") {
                callback(true, null)
            } else {
                callback(false, "Invalid email or password")
            }
        }.start()
    }
}
```

ViewModel — Exposes LiveData

```
class LoginViewModel(private val repository:
AuthRepository) : ViewModel() {

    val email = MutableLiveData("")
    val password = MutableLiveData("")
    val isLoading = MutableLiveData(false)
    val errorMessage = MutableLiveData<String?>()
    val loginSuccess = MutableLiveData(false)

    fun login() {
        val emailValue = email.value ?: ""
        val passwordValue = password.value ?: ""

        if (emailValue.isEmpty()) {
            errorMessage.value = "Email cannot be empty"
```

```
            return
        }
        if (passwordValue.isEmpty()) {
            errorMessage.value = "Password cannot be empty"
            return
        }

        isLoading.value = true

        repository.login(emailValue, passwordValue)
{ success, error ->
            isLoading.postValue(false)

            if (success) {
                loginSuccess.postValue(true)
            } else {
                errorMessage.postValue(error)
            }
        }
    }
}
```

View — Activity Using View Binding + Observers

```
class LoginActivity : AppCompatActivity() {

    private lateinit var vm: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)

        vm = ViewModelProvider(
            this,
            ViewModelProvider.Factory {
                LoginViewModel(AuthRepository())
            }
        )[LoginViewModel::class.java]

        val emailEdit = findViewById<EditText>(R.id.email)
        val passwordEdit =
findViewById<EditText>(R.id.password)
        val loginBtn = findViewById<Button>(R.id.login)
```

```
        val loading =
findViewById<ProgressBar>(R.id.progress)

        loginBtn.setOnClickListener { vm.login() }

        // Bind UI to ViewModel
        vm.email.observe(this) { emailEdit.setText(it) }
        vm.password.observe(this)
{ passwordEdit.setText(it) }
        vm.isLoading.observe(this) { loading.visibility = if
(it) View.VISIBLE else View.GONE }
        vm.errorMessage.observe(this) { it?.let
{ Toast.makeText(this, it, Toast.LENGTH_SHORT).show() } }
        vm.loginSuccess.observe(this) { if (it)
Toast.makeText(this, "Logged In!", Toast.LENGTH_SHORT).show()
}
    }
}
```

This is **real Android MVVM**:

- UI *observes* ViewModel
- ViewModel has **no reference** to the Activity
- No UI code in ViewModel

🔵 4. JavaScript MVVM Example (Vue.js)

Vue is literally built on MVVM.

Model

```
const api = {
  async login(email, password) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (email === "test@example.com" && password ===
"123456") {
          resolve({ id: 1, email });
        } else {
          reject("Invalid email or password");
        }
      }, 1000);
```

```
    });
  }
};
```

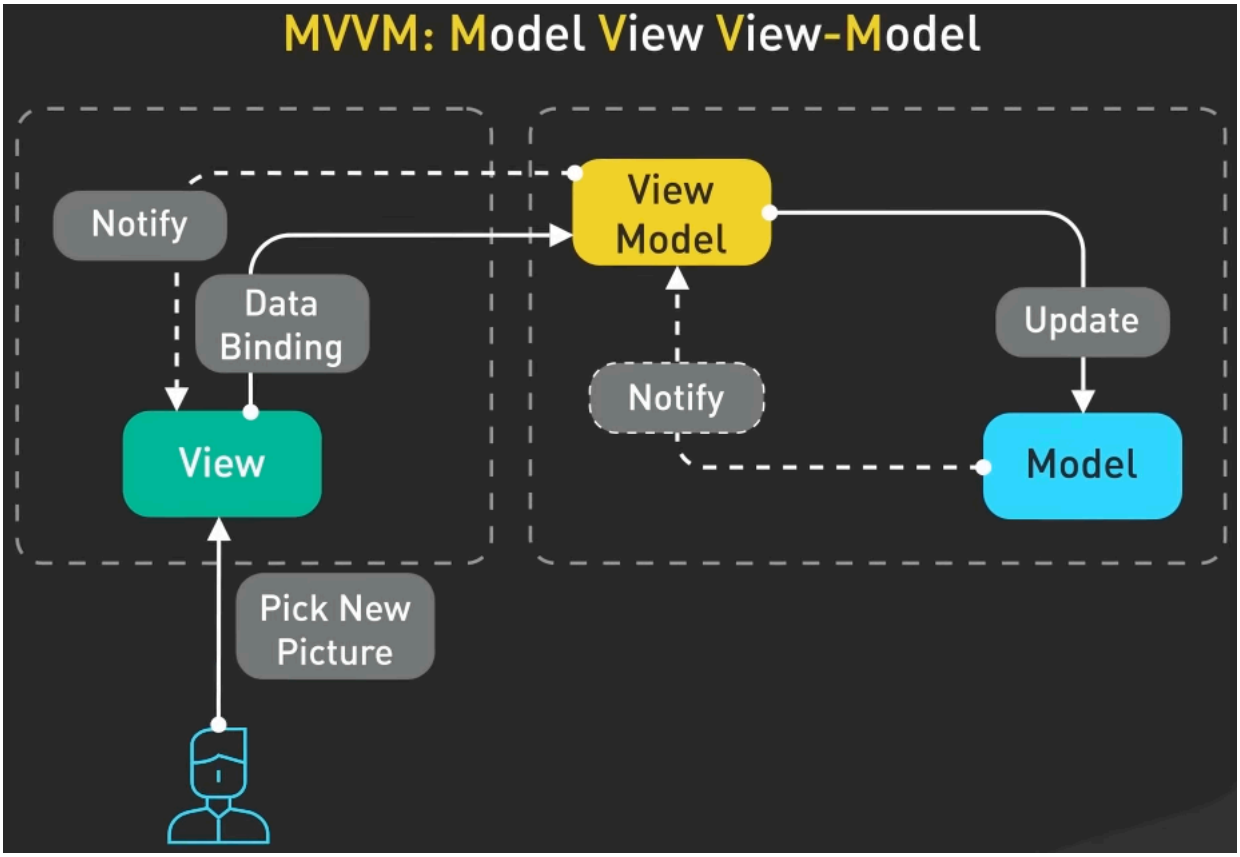
ViewModel (Vue component)

```
new Vue({
  el: "#app",

  data: {
    email: "",
    password: "",
    loading: false,
    errorMessage: "",
    success: false
  },

  methods: {
    async login() {
      this.errorMessage = "";
      this.loading = true;

      try {
        await api.login(this.email, this.password);
        this.success = true;
      } catch (err) {
        this.errorMessage = err;
      }

      this.loading = false;
    }
  }
});
```

View (HTML)

```
<div id="app">
  <input v-model="email" placeholder="Email" />
  <input v-model="password" type="password"
placeholder="Password" />

  <button @click="login" :disabled="loading">
    Login
```

```
  </button>

  <p v-if="loading">Loading...</p>
  <p v-if="errorMessage" style="color:red">{{ errorMessage }}
</p>
  <p v-if="success" style="color:green">Success!</p>
</div>
```



MVVM-C is basically **MVVM + a dedicated "navigation brain"** so your view controllers and view models don't have to know *how* to move between screens.

1. What is MVVM-C?

MVVM-C = Model – View – ViewModel – Coordinator

- **Model** – business logic, data, network, DB (same as usual)
- **View** – UIKit / SwiftUI UI code only (no business logic)
- **ViewModel** – exposes state & actions for the view, talks to Model
- **Coordinator** – owns:
    - Navigation flow (push, present, dismiss)
    - Creation & wiring of VCs + VMs
    - Flow transitions (e.g. after login → show main app)

In MVVM-C, **ViewModels do NOT push/present** view controllers.
Coordinators do it.

2. Why do people add the "C"?

Plain MVVM often gets messy because:

- ViewModel wants to trigger navigation:
    - It might call a delegate on the ViewController, or
    - It might need a reference to a Router / Navigator.
- ViewControllers end up:
    - Creating VMs
    - Knowing which screen comes next
    - Doing too much wiring

So MVVM-C introduces a **Coordinator** to:

- Centralize navigation
- Create VCs + VMs and inject dependencies
- Keep VMs and Views focused only on **screen logic**, not app flow.

3. iOS Example (Swift, UIKit) – Login → Home with MVVM-C

We'll build:

- A **Login screen**
- After success → navigate to a simple **Home screen**
- Using:
  - `AppCoordinator` – root of the app
  - `LoginCoordinator` – handles just login flow
  - `LoginViewModel`, `HomeViewModel`
  - `LoginViewController`, `HomeViewController`
  - `AuthService` (Model)

Folder structure (conceptual)

```
App/
  AppDelegate.swift
  SceneDelegate.swift
  Coordinators/
    AppCoordinator.swift
    LoginCoordinator.swift
  Models/
    User.swift
    AuthService.swift
  ViewModels/
    LoginViewModel.swift
    HomeViewModel.swift
  Views/
    LoginViewController.swift
    HomeViewController.swift
```

3.1 Model Layer

User.swift

```
struct User {
    let id: Int
```

```
    let email: String
}
```

AuthService.swift

```
import Foundation

protocol AuthServicing {
    func login(email: String,
               password: String,
               completion: @escaping (Result<User, Error>) ->
Void)
}

class AuthService: AuthServicing {

    func login(email: String,
               password: String,
               completion: @escaping (Result<User, Error>) ->
Void) {

        DispatchQueue.global().asyncAfter(deadline: .now() +
1.0) {
            if email == "test@example.com", password ==
"123456" {
                let user = User(id: 1, email: email)
                completion(.success(user))
            } else {
                let error = NSError(domain: "",
                                    code: 401,
                                    userInfo:
[NSLocalizedDescriptionKey: "Invalid credentials"])
                completion(.failure(error))
            }
        }
    }
}
```

3.2 Coordinator Protocol

A base coordinator type:

```
import UIKit

protocol Coordinator: AnyObject {
```

```
    var navigationController: UINavigationController { get }
    func start()
}
```

3.3 AppCoordinator – Entry point

Responsible for deciding the root flow (login vs main app). For now, always show login.

```
import UIKit

class AppCoordinator: Coordinator {

    let window: UIWindow
    let navigationController: UINavigationController
    private let authService: AuthServicing

    private var childCoordinators: [Coordinator] = []

    init(window: UIWindow) {
        self.window = window
        self.navigationController = UINavigationController()
        self.authService = AuthService()
    }

    func start() {
        window.rootViewController = navigationController
        window.makeKeyAndVisible()

        showLoginFlow()
    }

    private func showLoginFlow() {
        let loginCoordinator = LoginCoordinator(
            navigationController: navigationController,
            authService: authService
        )

        // When login finishes, show main app
        loginCoordinator.onLoginSuccess = { [weak self, weak
loginCoordinator] user in
            if let coordinator = loginCoordinator {
                self?.removeChild(coordinator)
            }
            self?.showHome(for: user)
        }
```

```
        childCoordinators.append(loginCoordinator)
        loginCoordinator.start()
    }

    private func showHome(for user: User) {
        let homeVM = HomeViewModel(user: user)
        let homeVC = HomeViewController(viewModel: homeVM)
        navigationController.setViewControllers([homeVC],
animated: true)
    }

    private func removeChild(_ coordinator: Coordinator) {
        childCoordinators.removeAll { $0 === coordinator }
    }
}
```

Notes:

- `AppCoordinator` holds window + nav controller
- Starts login flow via `LoginCoordinator`
- On success → `showHome(for:)`
- VMs and VCs never manipulate the nav stack directly.

3.4 LoginCoordinator – Handles Login Flow

```
import UIKit

class LoginCoordinator: Coordinator {

    let navigationController: UINavigationController
    private let authService: AuthServicing

    var onLoginSuccess: ((User) -> Void)?

    init(navigationController: UINavigationController,
         authService: AuthServicing) {
        self.navigationController = navigationController
        self.authService = authService
    }

    func start() {
```

```
        let viewModel = LoginViewModel(authService:
authService)

        viewModel.onLoginSuccess = { [weak self] user in
            self?.onLoginSuccess?(user)
        }

        let vc = LoginViewController(viewModel: viewModel)
        navigationController.setViewControllers([vc],
animated: false)
    }
}
```

Important:

- Coordinator creates `LoginViewModel` and `LoginViewController`.
- Coordinator wires the `onLoginSuccess` closure.

3.5 LoginViewModel – No UIKit, only logic + state

```
import Foundation

class LoginViewModel {

    // Inputs
    var email: String = ""
    var password: String = ""

    // Outputs (simple callbacks; could also use RxSwift/
Combine)
    var onLoadingStateChange: ((Bool) -> Void)?
    var onErrorMessage: ((String) -> Void)?
    var onLoginSuccess: ((User) -> Void)?

    private let authService: AuthServicing

    init(authService: AuthServicing) {
        self.authService = authService
    }

    func loginTapped() {
        guard !email.isEmpty else {
            onErrorMessage?("Email cannot be empty")
            return
```

```
        }

        guard !password.isEmpty else {
            onErrorMessage?("Password cannot be empty")
            return
        }

        onLoadingStateChange?(true)

        authService.login(email: email, password: password) {
[weak self] result in
            DispatchQueue.main.async {
                self?.onLoadingStateChange?(false)

                switch result {
                case .success(let user):
                    self?.onLoginSuccess?(user)
                case .failure(let error):
                    self?.onErrorMessage?
(error.localizedDescription)
                }
            }
        }
    }
}
```

Notes:

- No `UIViewController`, no `UINavigationController`.
- Emits state via closures.
- Coordinates only with Model & Coordinator (indirectly) through `onLoginSuccess`.

3.6 LoginViewController – View Layer

```
import UIKit

class LoginViewController: UIViewController {

    private let viewModel: LoginViewModel

    // UI elements
    private let emailField = UITextField()
    private let passwordField = UITextField()
```

```
    private let loginButton = UIButton(type: .system)
    private let activityIndicator =
UIActivityIndicatorView(style: .medium)

    init(viewModel: LoginViewModel) {
        self.viewModel = viewModel
        super.init(nibName: nil, bundle: nil)
        title = "Login"
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = .systemBackground
        setupUI()
        bindViewModel()
    }

    private func setupUI() {
        emailField.placeholder = "Email"
        emailField.borderStyle = .roundedRect
        emailField.keyboardType = .emailAddress
        emailField.autocapitalizationType = .none

        passwordField.placeholder = "Password"
        passwordField.borderStyle = .roundedRect
        passwordField.isSecureTextEntry = true

        loginButton.setTitle("Login", for: .normal)

        let stack = UIStackView(arrangedSubviews:
[emailField, passwordField, loginButton, activityIndicator])
        stack.axis = .vertical
        stack.spacing = 16

        stack.translatesAutoresizingMaskIntoConstraints =
false
        view.addSubview(stack)

        NSLayoutConstraint.activate([
            stack.centerYAnchor.constraint(equalTo:
view.centerYAnchor),
```

```
            stack.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 24),
            stack.trailingAnchor.constraint(equalTo:
view.trailingAnchor, constant: -24)
        ])

        loginButton.addTarget(self, action:
#selector(loginButtonTapped), for: .touchUpInside)
    }

    private func bindViewModel() {
        viewModel.onLoadingStateChange = { [weak self]
isLoading in
            self?.activityIndicator.isHidden = !isLoading
            if isLoading {
                self?.activityIndicator.startAnimating()
                self?.loginButton.isEnabled = false
            } else {
                self?.activityIndicator.stopAnimating()
                self?.loginButton.isEnabled = true
            }
        }

        viewModel.onErrorMessage = { [weak self] message in
            let alert = UIAlertController(title: "Error",
                                          message: message,

preferredStyle: .alert)
            alert.addAction(UIAlertAction(title: "OK", style:
.default))
            self?.present(alert, animated: true)
        }
    }

    @objc private func loginButtonTapped() {
        viewModel.email = emailField.text ?? ""
        viewModel.password = passwordField.text ?? ""
        viewModel.loginTapped()
    }
}
```

Notes:

- ViewController does **no navigation**.
- Just binds UI to ViewModel callbacks & triggers actions.

3.7 HomeViewModel & HomeViewController

HomeViewModel.swift

```
class HomeViewModel {
    let welcomeText: String

    init(user: User) {
        self.welcomeText = "Welcome, \(user.email)"
    }
}
```

HomeViewController.swift

```
import UIKit

class HomeViewController: UIViewController {

    private let viewModel: HomeViewModel
    private let label = UILabel()

    init(viewModel: HomeViewModel) {
        self.viewModel = viewModel
        super.init(nibName: nil, bundle: nil)
        title = "Home"
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = .systemBackground

        label.text = viewModel.welcomeText
        label.textAlignment = .center

        label.translatesAutoresizingMaskIntoConstraints =
false
        view.addSubview(label)

        NSLayoutConstraint.activate([
```

```
            label.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            label.centerYAnchor.constraint(equalTo:
view.centerYAnchor)
        ])
    }
}
```

3.8 AppDelegate / SceneDelegate Wiring

SceneDelegate.swift (simplified)

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?
    var appCoordinator: AppCoordinator?

    func scene(_ scene: UIScene,
               willConnectTo session: UISceneSession,
               options connectionOptions:
UIScene.ConnectionOptions) {

        guard let windowScene = (scene as? UIWindowScene)
else { return }

        let window = UIWindow(windowScene: windowScene)
        self.window = window

        let appCoordinator = AppCoordinator(window: window)
        self.appCoordinator = appCoordinator
        appCoordinator.start()
    }
}
```

4. How MVVM-C Flows (Login Case)

1. **App start**
    - `SceneDelegate` → creates `AppCoordinator` → `start()`
    - `AppCoordinator` shows `LoginCoordinator`
2. **Login flow**
    - `LoginCoordinator` creates:

        - `LoginViewModel(authService:)`
        - `LoginViewController(viewModel:)`
    - Pushes `LoginViewController` on nav stack

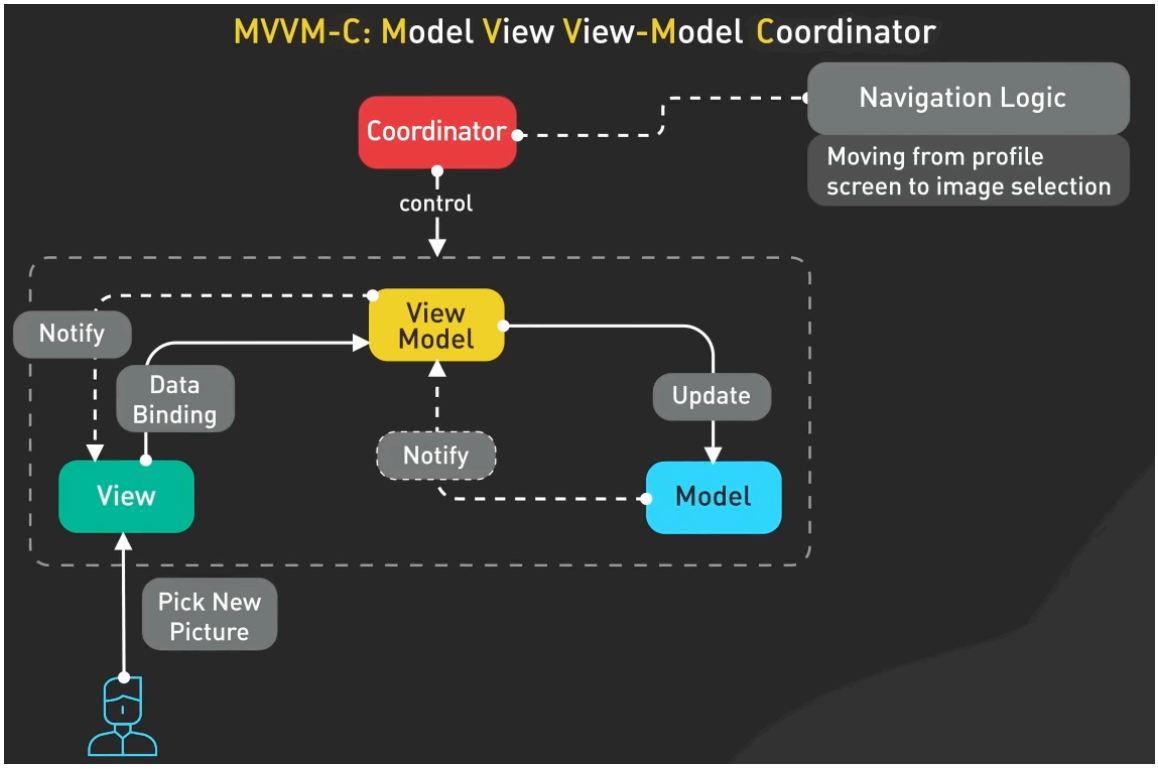
3. **User taps login**
    - `LoginViewController` → `viewModel.loginTapped()`
    - `LoginViewModel` → calls `authService.login(...)`
    - On success → calls `onLoginSuccess(user)`

4. **Coordinator reacts**
    - `LoginCoordinator`'s `onLoginSuccess` triggers `AppCoordinator.showHome(for:)`
    - `AppCoordinator` sets nav root to `HomeViewController`

At no point does the ViewModel or View do `navigationController.pushViewController`.
All navigation is in Coordinators.

MVVM-C is especially nice when:

- You have **many flows** (onboarding, login, main, settings, etc.)
- You want **ViewModels and Views to be navigation-agnostic**
- You want **testable flows** and **centralized wiring/injection**



VIPER is like MVVM-C on steroids: **even more layers, even stricter separation**.

1. What **VIPER** is (concept + roles)
2. How data & events flow in VIPER
3. A **full iOS Swift VIPER Login module**
    - Entity
    - Interactor
    - Presenter
    - View (UIViewController)
    - Router (aka Wireframe)
    - Module builder

1. What is VIPER?

VIPER = View – Interactor – Presenter – Entity – Router

It's a "Clean Architecture"-style pattern used a lot in iOS projects to:

- Make each screen **highly modular**
- Push business logic into Interactors (use cases)
- Keep ViewControllers super thin
- Centralize navigation in Router
- Make everything very **testable** and **replaceable**

VIPER Roles (High Level)

1. **View**
    - UIKit / SwiftUI layer
    - Shows data, receives user actions
    - Talks only to the Presenter via a protocol
    - No business logic
2. **Presenter**
    - Middleman / coordinator for one screen

- Receives user actions from View
- Asks Interactor to perform work
- Formats data for the View
- Asks Router to navigate
- Contains no UIKit types

3. **Interactor**
   - Contains **business logic** and **use cases**
   - Talks to services / repositories / APIs
   - Returns raw domain data (Entities) back to Presenter
4. **Entity**
   - Simple data models (User, Product, etc.)
   - Pure Swift structs/classes that represent business objects
5. **Router (Wireframe)**
   - Responsible for **navigation & module creation**
   - Push / present / dismiss ViewControllers
   - Knows how to build a VIPER module (wire all dependencies together)

A typical rule:

- **View ↔ Presenter**
- **Presenter ↔ Interactor**
- **Presenter ↔ Router**
- **Interactor ↔ Entities / Services**

2. VIPER Flow (Conceptual)

Let's use a **Login screen** as example:

1. User taps **Login**
2. View tells Presenter: `presenter.didTapLogin(email: password:)`
3. Presenter validates input (optionally) and tells Interactor:
   `interactor.login(email: password:)`

4. Interactor calls `AuthService`, gets `User` or error
5. Interactor reports back to Presenter:
    - `loginSucceeded(user: User)` or
    - `loginFailed(error: Error)`
6. Presenter formats message and:
    - tells View to show error OR
    - tells Router to navigate to the Home screen

At no point does:

- The View call services directly
- The Interactor talk to UIKit
- The Presenter push ViewControllers itself

3. iOS Swift VIPER Example — Login Module

We'll create a **LoginModule** with:

- `LoginEntity.swift` (User)
- `AuthService.swift`
- `LoginProtocols.swift`
- `LoginInteractor.swift`
- `LoginPresenter.swift`
- `LoginRouter.swift`
- `LoginViewController.swift`
- `LoginModuleBuilder` (static factory)

3.1 Entity & Service (Model Layer)

User.swift

```
struct User {
    let id: Int
```

```
    let email: String
}
```

AuthService.swift

```
import Foundation

protocol AuthServicing {
    func login(email: String,
               password: String,
               completion: @escaping (Result<User, Error>) ->
Void)
}

class AuthService: AuthServicing {
    func login(email: String,
               password: String,
               completion: @escaping (Result<User, Error>) ->
Void) {

        DispatchQueue.global().asyncAfter(deadline: .now() +
1.0) {
            if email == "test@example.com", password ==
"123456" {
                completion(.success(User(id: 1, email:
email)))
            } else {
                let error = NSError(domain: "",
                                    code: 401,
                                    userInfo:
[NSLocalizedDescriptionKey: "Invalid email or password"])
                completion(.failure(error))
            }
        }
    }
}
```

3.2 VIPER Protocols (Contracts)

We define protocols so everything can be mocked & decoupled.

LoginProtocols.swift

```
import UIKit
```

```
// MARK: - View

protocol LoginViewProtocol: AnyObject {
    var presenter: LoginPresenterProtocol? { get set }

    func showLoading(_ isLoading: Bool)
    func showError(_ message: String)
    func clearError()
}

// MARK: - Presenter

protocol LoginPresenterProtocol: AnyObject {
    func viewDidLoad()
    func didTapLogin(email: String?, password: String?)
}

// MARK: - Interactor

protocol LoginInteractorInputProtocol: AnyObject {
    func login(email: String, password: String)
}

protocol LoginInteractorOutputProtocol: AnyObject {
    func loginSucceeded(user: User)
    func loginFailed(error: Error)
}

// MARK: - Router

protocol LoginRouterProtocol: AnyObject {
    static func createModule() -> UIViewController
    func navigateToHome(from view: LoginViewProtocol, user:
User)
}
```

Notes:

- `View` has a `presenter` reference
- Presenter talks to:
    - View via `LoginViewProtocol`
    - Interactor via `LoginInteractorInputProtocol`
    - Router via `LoginRouterProtocol`

- Interactor reports back via `LoginInteractorOutputProtocol`

3.3 Interactor — Business Logic

LoginInteractor.swift

```
import Foundation

class LoginInteractor: LoginInteractorInputProtocol {

    weak var presenter: LoginInteractorOutputProtocol?
    private let authService: AuthServicing

    init(authService: AuthServicing) {
        self.authService = authService
    }

    func login(email: String, password: String) {
        authService.login(email: email, password: password) {
[weak self] result in
            DispatchQueue.main.async {
                switch result {
                case .success(let user):
                    self?.presenter?.loginSucceeded(user:
user)
                case .failure(let error):
                    self?.presenter?.loginFailed(error:
error)
                }
            }
        }
    }
}
```

- No UIKit
- Only talks to `AuthService` + `Presenter` output

3.4 Presenter — Orchestrator / Middleman

LoginPresenter.swift

```
import Foundation
```

```
class LoginPresenter: LoginPresenterProtocol {

    weak var view: LoginViewProtocol?
    var interactor: LoginInteractorInputProtocol?
    var router: LoginRouterProtocol?

    func viewDidLoad() {
        // Any initial setup if needed
        view?.clearError()
    }

    func didTapLogin(email: String?, password: String?) {
        view?.clearError()

        guard let email = email, !email.isEmpty else {
            view?.showError("Email cannot be empty")
            return
        }
        guard let password = password, !password.isEmpty else
{
            view?.showError("Password cannot be empty")
            return
        }

        view?.showLoading(true)
        interactor?.login(email: email, password: password)
    }
}

// MARK: - Interactor Output

extension LoginPresenter: LoginInteractorOutputProtocol {

    func loginSucceeded(user: User) {
        view?.showLoading(false)
        // Ask Router to navigate
        if let view = view {
            router?.navigateToHome(from: view, user: user)
        }
    }

    func loginFailed(error: Error) {
        view?.showLoading(false)
        view?.showError(error.localizedDescription)
    }
```

```
}
```
Key points:

- Validates input
- Calls Interactor
- Handles success/failure and asks Router to navigate
- No navigation code itself

3.5 Router — Navigation & Module Creation

LoginRouter.swift

```
import UIKit

class LoginRouter: LoginRouterProtocol {

    static func createModule() -> UIViewController {
        let view = LoginViewController()

        let presenter = LoginPresenter()
        let authService = AuthService()
        let interactor = LoginInteractor(authService:
authService)
        let router = LoginRouter()

        // Wiring VIPER components
        view.presenter = presenter

        presenter.view = view
        presenter.interactor = interactor
        presenter.router = router

        interactor.presenter = presenter

        return view
    }

    func navigateToHome(from view: LoginViewProtocol, user:
User) {
        guard let sourceVC = view as? UIViewController else {
return }
```

```
        let homeVC = HomeViewController(user: user)

sourceVC.navigationController?.setViewControllers([homeVC],
animated: true)
    }
}
```

This is where **wiring + injection** happen:

- Router builds the whole module
- Injects dependencies into Presenter and Interactor
- Responsible for navigation

3.6 View — UIKit Layer (LoginViewController)

LoginViewController.swift

```
import UIKit

class LoginViewController: UIViewController,
LoginViewProtocol {

    var presenter: LoginPresenterProtocol?

    // UI
    private let emailField = UITextField()
    private let passwordField = UITextField()
    private let loginButton = UIButton(type: .system)
    private let errorLabel = UILabel()
    private let activityIndicator =
UIActivityIndicatorView(style: .medium)

    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = .systemBackground
        title = "Login"

        setupUI()
        presenter?.viewDidLoad()
    }

    private func setupUI() {
        emailField.placeholder = "Email"
        emailField.borderStyle = .roundedRect
```

```
        emailField.keyboardType = .emailAddress
        emailField.autocapitalizationType = .none

        passwordField.placeholder = "Password"
        passwordField.borderStyle = .roundedRect
        passwordField.isSecureTextEntry = true

        loginButton.setTitle("Login", for: .normal)
        loginButton.addTarget(self, action:
#selector(loginTapped), for: .touchUpInside)

        errorLabel.textColor = .systemRed
        errorLabel.numberOfLines = 0
        errorLabel.textAlignment = .center

        activityIndicator.hidesWhenStopped = true

        let stack = UIStackView(arrangedSubviews:
[emailField, passwordField, loginButton, activityIndicator,
errorLabel])
        stack.axis = .vertical
        stack.spacing = 16
        stack.translatesAutoresizingMaskIntoConstraints =
false

        view.addSubview(stack)
        NSLayoutConstraint.activate([
            stack.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 24),
            stack.trailingAnchor.constraint(equalTo:
view.trailingAnchor, constant: -24),
            stack.centerYAnchor.constraint(equalTo:
view.centerYAnchor)
        ])
    }

    @objc private func loginTapped() {
        presenter?.didTapLogin(email: emailField.text,
                               password: passwordField.text)
    }

    // MARK: - LoginViewProtocol

    func showLoading(_ isLoading: Bool) {
        if isLoading {
```

```
            activityIndicator.startAnimating()
            loginButton.isEnabled = false
        } else {
            activityIndicator.stopAnimating()
            loginButton.isEnabled = true
        }
    }

    func showError(_ message: String) {
        errorLabel.text = message
    }

    func clearError() {
        errorLabel.text = ""
    }
}
```

View is **thin**:

- No logic about AuthService
- No navigation code
- Just sends user events to Presenter and renders state

3.7 Home Screen (Simple Non-VIPER for Demo)

HomeViewController.swift

```
import UIKit

class HomeViewController: UIViewController {

    private let user: User
    private let label = UILabel()

    init(user: User) {
        self.user = user
        super.init(nibName: nil, bundle: nil)
        title = "Home"
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = .systemBackground

        label.text = "Welcome, \(user.email)"
        label.textAlignment = .center
        label.translatesAutoresizingMaskIntoConstraints =
false

        view.addSubview(label)
        NSLayoutConstraint.activate([
            label.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            label.centerYAnchor.constraint(equalTo:
view.centerYAnchor)
        ])
    }
}
```

3.8 App Entry (e.g. SceneDelegate)

You'd start the module like:

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(_ scene: UIScene,
               willConnectTo session: UISceneSession,
               options connectionOptions:
UIScene.ConnectionOptions) {

        guard let windowScene = (scene as? UIWindowScene)
else { return }

        let window = UIWindow(windowScene: windowScene)
        let rootVC = LoginRouter.createModule()
        let nav = UINavigationController(rootViewController:
rootVC)
        window.rootViewController = nav
        window.makeKeyAndVisible()
        self.window = window
    }
}
```

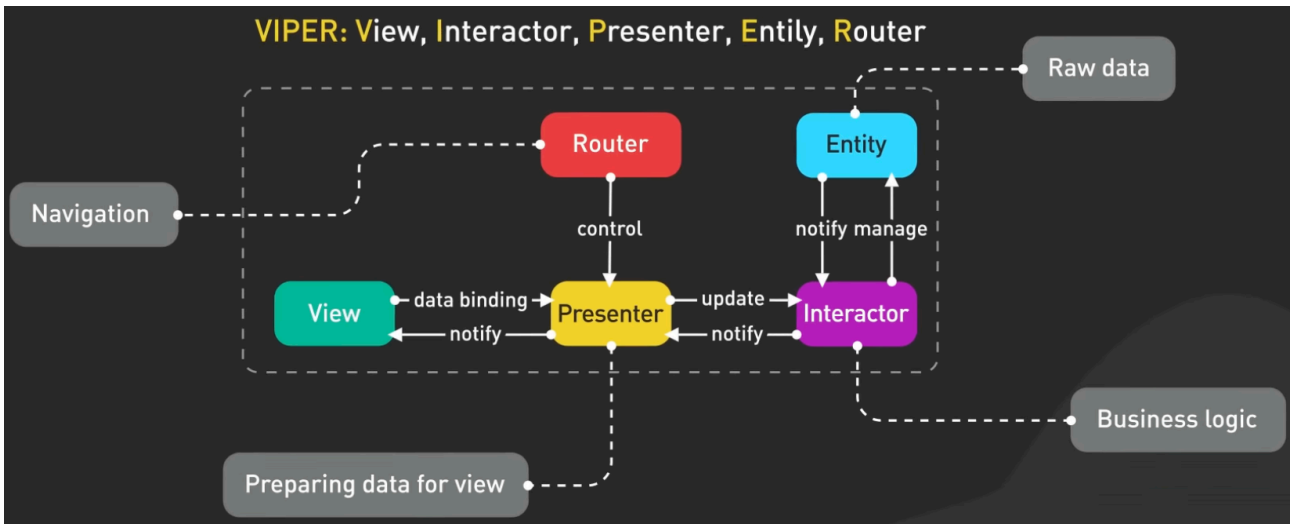
4. VIPER Data / Responsibility Summary

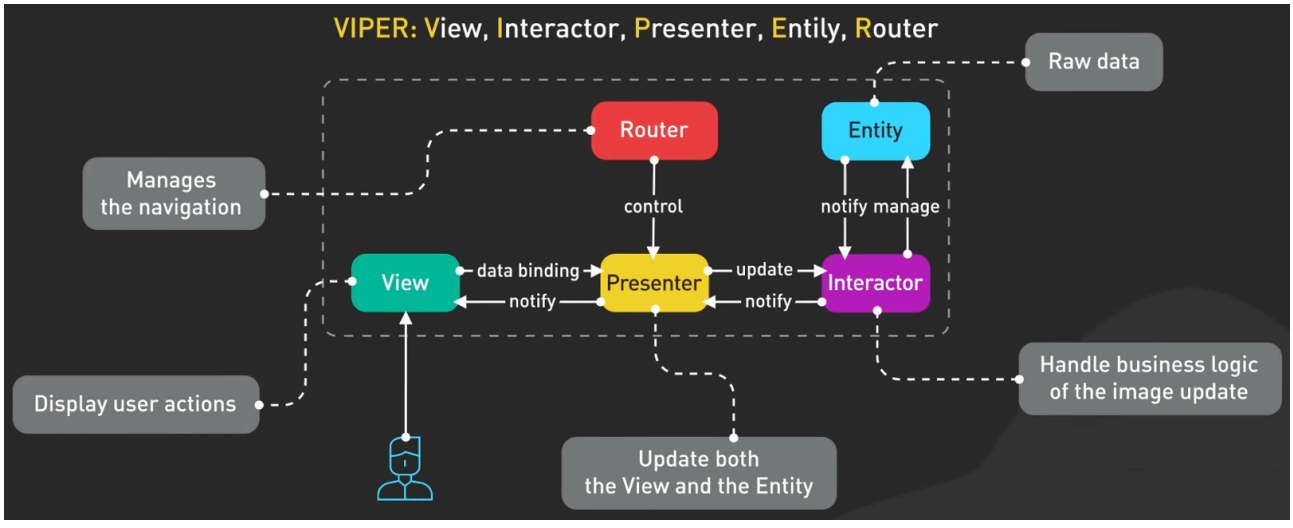
Data & Events:

- View → Presenter: user events
- Presenter → Interactor: business actions
- Interactor → Presenter: results / entities
- Presenter → View: UI state changes
- Presenter → Router: navigation requests
- Router → ViewControllers: actual UIKit navigation

Benefits:

- Super clear **separation of concerns**
- Very **testable** (Presenter & Interactor are pure Swift)
- Easy to maintain big apps with complex flows





⭐Architecture Comparison Table (MVC / MVP / MVVM / MVVM-C / VIPER)

Category	MVC	MVP	MVVM	MVVM-C	VIPER
Full Name	Model-View-Controller	Model-View-Presenter	Model-View-ViewModel	MVVM + Coordinator	View-Interactor-Presenter-Entity-Router
Primary Goal	Separate UI & business logic	Make View passive, Presenter active	Make UI reactive with bindable state	Add scalable navigation to MVVM	Strict Clean Architecture, high testability
View Responsibiliti	Heavy; often handles logic	Passive; only UI & events	Reactive UI; listens to	Same as MVVM	Very thin UI only
Controller / Presenter / ViewModel role	Controller handles UI + navigation	Presenter handles UI logic; View is dumb	ViewModel provides observable state; no UI code	Same ViewModel, but no navigation inside it	Presenter orchestrates View + Interactor
Navigation Lives In	Controller (bad for large apps)	View or Presenter (varies)	Often View or ViewModel (messy)	**Coordinator (centralized navigation)**	**Router (strict navigation layer)**
Business Logic Lives In	Controller or Model (mixed)	Presenter + Model	ViewModel + Model	ViewModel + Interactor (optional)	**Interactor** (pure use cases)
Data Binding	❌ (manual updates)	❌	✔ Reactive (Combine/Rx/LiveData)	✔ Reactive	❌ (Presenter manually updates View)
Testability	Low–Medium	Medium–High	High	Very High	Very High (Interactor + Presenter isolation)
Object Creation	Often inside Controller	Often inside View or	Sometimes inside View	**Coordinator wires**	**Router wires everything**
Risk of Massive Classes	“Massive ViewController” is	Presenter can get big	ViewModel can get big	Coordinator fixes ViewModel size	Almost none if done correctly
Best For	Simple apps; prototypes	Medium apps; when you want a passive View	Modern UI frameworks; reactive UIs	Medium–large apps; flows & navigation	Large enterprises; long-term scalable apps
Layers	3	3	3	4	5
iOS Popularity	Old UIKit days	Some usage	Very popular (SwiftUI)	Very popular in UIKit	Common in enterprise teams
Android Popularity	Rare	Old Android days	Jetpack standard	Sometimes (using	Rare
Communication Flow	View ↔ Controller ↔ Model	View → Presenter ↔ Model	View ↔ ViewModel ↔ Model	View ↔ ViewModel ↔ Coordinator ↔ Model	View → Presenter ↔ Interactor ↔ Entities

Who Holds the App Flow	Controller	Presenter or View	View/ViewModel	**Coordinator**	**Router**

⭐ Summary in One Sentence Each

MVC:

Fastest to build but becomes messy; Controller becomes God-object.

MVP:

View is dumb, Presenter is smart; easy to test; but navigation can be scattered.

MVVM:

Reactive UI + ViewModel state; clean but navigation unclear without extra patterns.

MVVM-C:

MVVM + Coordinator = beautiful, scalable flows; ideal for iOS UIKit architecture.

VIPER:

Most strict and modular; enterprise-level separation; lots of boilerplate but very clean long-term.