MVC Architecture

Interview Answers

3DaysOfSwift.com

Swift Fundamentals. Interview Ready.

MVC Architecture Interview Questions

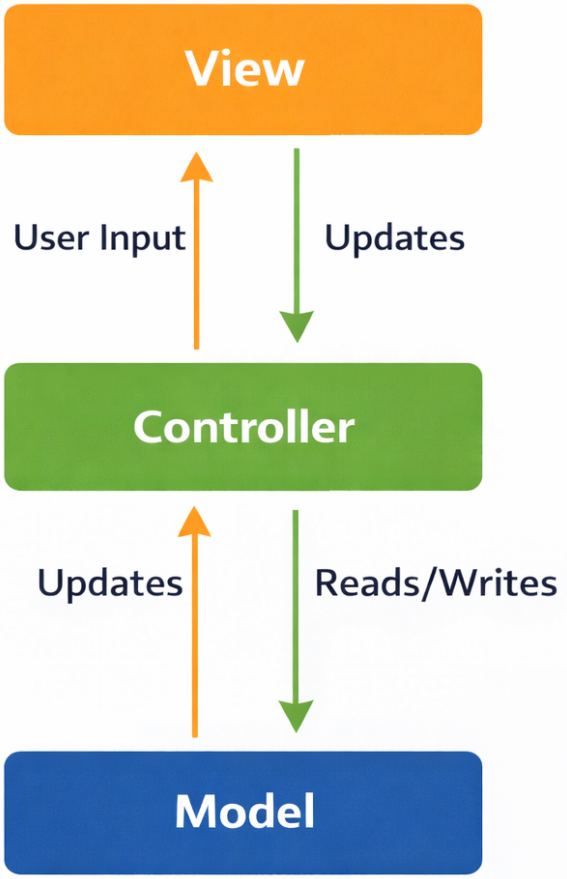
for Junior iOS Developers

V1 ©copyright of 3DaysOfSwift.com

What Is MVC?

MVC stands for **Model–View–Controller**:

- the **Model** holds the data and rules
- the **View** shows things on screen
- and the **Controller** connects the two, controlling the view based on the *state* of the model



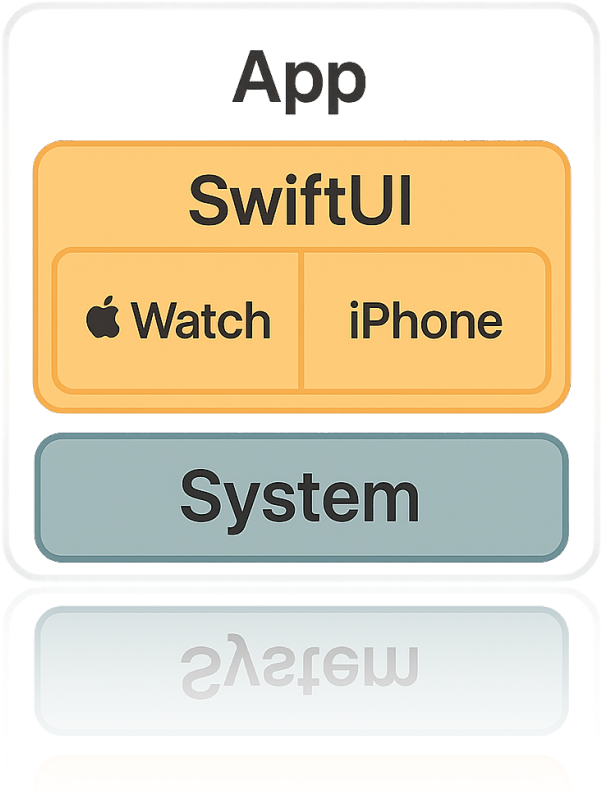
In iOS apps, 📲

View - the user taps buttons and types into text fields (input).

Controller - The Controller reacts to that input, asks the Model for data or updates it, and then tells the View what to display.

Model - Isolated underlying system that can be freely editted, updated and maintained without affecting the main view of the app.

The Model is the brain 🧠 of the operation and is the main underlying system.



The basic premis of MVC is to separate the UI code from the underlying system beneath it.

If we want our Model (the system) to run on a different device (such as Watch) then we can write new SwiftUI files to read and interact with it.

This isolates the system, preventing changes from occuring unintentionally or by mistake.

Interview Questions

What is MVC?

MVC stands for Model–View–Controller. It separates data and business rules (Model), user interface (View), and coordination and user input handling (Controller). The goal is clearer responsibilities, easier maintenance, and simpler testing.

What is a Model?

A Model represents data and business rules. It contains core logic and state that should remain valid regardless of how the data is displayed or which UI framework is used.

What is a View?

A View is responsible for displaying information to the user. It should contain no business logic and should only reflect the state it is given.

What is a Controller?

A Controller coordinates between the View and the Model. It responds to user input, requests data from the model, and updates the view accordingly.

What is MVVM?

MVVM stands for Model–View–ViewModel. It introduces a ViewModel to hold presentation logic and UI-ready state, reducing the responsibility of the view and improving testability.

What is Layered Architecture?

Layered architecture organises code into layers such as Presentation, Business Logic, and Data. Each layer has a clear responsibility and communicates only with neighbouring layers.

What are the SOLID principles?

SOLID is a set of five design principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. Together they encourage flexible, maintainable, and testable object-oriented code.

What is Single Responsibility?

The Single Responsibility Principle states that a type should have only one reason to change. This keeps code easier to understand, test, and modify safely.

What is dependency injection?

Dependency injection is the practice of supplying dependencies from the outside rather than creating them internally. It reduces coupling and makes code easier to test, replace, and extend.

What is Protocol-Oriented Programming (POP)?

Protocol-Oriented Programming is a Swift paradigm that emphasises behaviour defined in protocols rather than inheritance. It promotes composition, flexibility, and code reuse.

What is Copy-on-Write in Swift?

Copy-on-Write is a performance optimisation where value types share memory until one is mutated. Swift collections like Array and Dictionary use this to remain efficient while preserving value semantics.

3DaysOfSwift.com

Swift Fundamentals. Interview Ready.

What is declarative vs imperative programming?

Imperative programming describes how to perform tasks step by step, while declarative programming describes what the desired outcome is. SwiftUI is declarative, whereas UIKit is largely imperative.

What is the Combine framework?

Combine is Apple's reactive framework for handling asynchronous events and data streams. It allows developers to model values that change over time using publishers and subscribers.

Why is Combine useful with SwiftUI?

Combine integrates naturally with SwiftUI by driving UI updates from reactive data streams. This results in predictable state changes and less manual UI synchronisation.

What is KISS?

KISS stands for Keep It Simple, Stupid. It encourages writing the simplest solution that solves the problem, reducing complexity and long-term maintenance costs.

3DaysOfSwift.com

Swift Fundamentals. Interview Ready.

Why build code using components rather than huge files?

Small components improve readability, reuse, and testability. They reduce cognitive load and limit the impact of changes to isolated areas of the codebase.

What makes code testable?

Testable code is isolated, deterministic, and free from UI or framework dependencies. It relies on clear inputs, outputs, and injected dependencies.

What is separation of concerns?

Separation of concerns means each component has a single responsibility and reason to change. This keeps systems easier to understand and modify.

Can architecture be over-engineered?

Yes. Architecture should match the size and complexity of the problem. Over-engineering increases cognitive load and slows development.

How do architectural decisions affect long-term maintenance?

Good architecture reduces refactoring cost, improves onboarding speed, and limits the impact of changes as systems grow.

How do architectural decisions help teams?

Clear architecture enables parallel work, reduces merge conflicts, and clarifies ownership across teams.

So, Let's Clarify..

What is MVC?

MVC stands for Model–View–Controller, and the simplest way to remember it is: the Model holds the data and rules, the View shows things on screen, and the Controller connects the two. In an iOS app, the user taps buttons and types into text fields (input). The Controller reacts to that input, asks the Model for data or updates it, and then tells the View what to display. The benefit is that you can change how the UI looks (View) without rewriting your core logic (Model), and you can test the Model logic without needing the UI running.

What is a Model?

A Model is the part of your code that represents real information and the rules around it. For example, a User model might contain a user's name, email, and permissions; a ShoppingCart model might contain items and a method to calculate the total price. A good Model does not know anything about buttons, screens, or colours. It should be reusable even if you changed your app from iPhone to web or to an API, because the rules and data stay the same.

What is a View?

A View is responsible for presenting information to the user. In UIKit, this could be labels, buttons, table views, and custom UIViews. In SwiftUI, the View is the struct that describes what should appear on screen. A View should avoid business rules like 'is this user allowed to purchase?'—it should simply display whatever state it receives. Keeping Views 'dumb' makes UI code easier to change without breaking the rest of the system.

What is a Controller?

A Controller is the coordinator between the View and the Model. In UIKit, the most common Controller is a UIViewController. It listens for user actions (like tapping a button), triggers work (like asking a service to load data), and then updates what the user sees (like reloading a table). Controllers often become too large if you let them own business logic, networking, or complex state—so a good habit is to keep controllers focused on orchestration rather than heavy logic.

What is MVVM?

MVVM stands for Model–View–ViewModel. It still has Models and Views, but adds a ViewModel that prepares data specifically for the screen. Think of the ViewModel as the place where you transform raw model data into 'display-ready' values, like turning a Date into a formatted string, or combining first and last name into a full name. This helps keep the View and ViewController simpler, and it makes the presentation logic easier to unit test.

What is Layered Architecture?

Layered architecture is a broader way of organising code into layers such as Presentation (UI), Domain/Business (rules), and Data (networking, database). The idea is that each layer has a clear job: the UI layer handles display and user interaction, the business layer decides what should happen, and the data layer knows how to fetch or save information. When these layers are respected, your codebase is easier to navigate, and you can change one layer (like swapping a database) without rewriting everything.

Why separate the presentation layer from the underlying system?

The presentation layer changes frequently: designs change, layouts change, and you might build the same product for iOS, Android, and web. If your business rules are mixed into the UI, every UI change risks breaking core logic. Separating them means your 'engine' stays stable while the UI can evolve. It also improves testing, because you can test business logic without needing to run a UI or simulate taps.

Why build code using components rather than huge files?

Large files usually mean too many responsibilities in one place, which makes bugs harder to find and changes riskier. Components are small, focused pieces of code that do one job well—like a networking client, a formatter, a validator, or a reusable UI component. This makes code easier to read, easier to reuse, and easier to test. It also helps teams work in parallel, because different developers can own different components without constantly conflicting in the same file.

What is a Massive View Controller and why does it happen?

A ‘Massive View Controller’ happens when a UIViewController slowly collects responsibilities: UI code, business rules, networking calls, parsing, navigation, and state management. It happens because it feels convenient at first—everything is in one place—but over time the file becomes hard to understand and risky to change. The fix is to move work into smaller components: put networking into services, business rules into models/use-cases, formatting into helpers, and presentation logic into a ViewModel or similar structure.