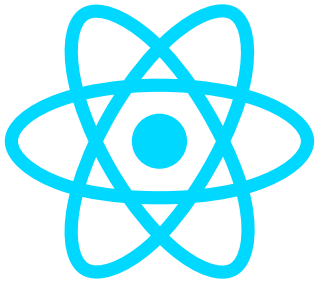

React Native Developer Interview Handbook

Your Ultimate Guide to Crack React Native Interviews with Confidence

500+ Real Interview Questions with Clear Explanations

Written By
Anand Gaur

Table of Contents

1. Introduction

Why This Handbook?

Preparing for React Native interviews can feel confusing because the expectations vary widely depending on the company, role, and project type. One round may focus heavily on JavaScript or TypeScript fundamentals, another may test your understanding of React, hooks, lifecycle, or performance. At senior levels, you'll often be asked about architecture, native modules, bridging, large-scale app design, and how you optimize cross-platform performance.

That's exactly why this handbook was created to be your one-stop guide for React Native interview preparation. Instead of learning from scattered YouTube videos, blogs, GitHub repos, and outdated tutorials, this book puts everything together in a structured format.

As a developer, I've personally experienced how difficult it is to prepare for React Native interviews. You end up juggling between React docs, JavaScript guides, native platform notes, and multiple community libraries. There wasn't a single resource that combined real interview questions, clean explanations, practical examples, and scenario-based discussions so I decided to build one.

This handbook is the result of years of interview experience, combined with questions collected from real interviews across product companies, startups, and service-based organizations. I documented every challenging question I came across not just to improve myself, but to help developers prepare smarter, faster, and with confidence.

Inside this handbook, you'll find:

- Topic-wise categorized questions covering everything: JavaScript, TypeScript, React concepts, React Native APIs, native modules, performance, architecture, DevOps, and more
- Coverage for all levels — from freshers starting with React basics to senior developers and leads preparing for design and architecture rounds
- Scenario-based discussions that reflect real-world interview patterns
- Latest React Native ecosystem topics like Fabric Architecture, TurboModules, Hermes, Reanimated, state management trends, and cross-platform optimizations

No matter your experience level, this handbook will give you a clear roadmap of what to expect and how to prepare so you walk into your interview confident and well-prepared.

How to Use This Handbook Effectively

Think of this handbook as a structured preparation guide instead of random notes. Here's the ideal order:

1. Start with fundamentals

If you're a fresher, begin with JavaScript and TypeScript basics, ES6 concepts, React fundamentals, JSX, components, and the React Native lifecycle.

2. Move into UI development

Learn styling, Flexbox, FlatList optimizations, navigation (React Navigation), gestures, animations, and platform-specific UI behavior.

3. Deep-dive into state management

Practice with Context API, Redux, Redux Toolkit, Zustand, MobX, Recoil, and understand when to choose which.

4. Explore data handling and async programming

Networking with fetch/axios, caching, offline mode, async/await, parallel API calls, promises, WebSockets.

5. Grow into advanced concepts

React Native architecture, Fabric, TurboModules, native modules using Swift/Kotlin, performance tuning, bundle optimization.

6. Don't ignore DevOps and Security

App builds, release processes (Play Store, App Store), CI/CD, OTA updates (CodePush), SSL pinning, secure storage, crash analytics.

7. Wrap up with scenario-based questions

These sharpen your real-world thinking and prepare you for problem-solving rounds like:

- How do you optimize a complex FlatList?
- How do you manage state in a large-scale app?
- How would you design offline-first data flow?
- How to integrate a native Android/iOS SDK in RN?

Following this sequence ensures you build a complete understanding instead of studying random topics without direction.

Levels of React Native Interviews

Fresher / Junior (0–2 years)

Expect questions on:

- JavaScript basics, ES6, async/await
- React fundamentals: components, props, state, hooks
- React Native basics: views, styling, navigation, API calls
- FlatList, basic forms, error handling
- Differences between React and React Native

Mid-level (2–5 years)

Focus shifts to:

- Performance optimization concepts
- State management (Redux, Context, Zustand)
- React Navigation internals
- Local storage (AsyncStorage, MMKV, SQLite)
- Native APIs, lifecycle, animations
- Debugging and profiling
- TypeScript best practices

Senior (5+ years)

You'll be evaluated on:

- Designing scalable apps and modular architectures
- Clean architecture patterns in RN
- Native module creation and bridging
- Performance optimization (Hermes, re-renders, batching, virtualization)
- CI/CD pipelines, OTA updates (CodePush)
- Leading teams, mentoring, and reviewing code
- Handling large, cross-functional projects

Architect / Lead

Interviews go far beyond coding:

- Enterprise-level architecture, micro-frontends, monorepos
- Large-scale state management strategies
- Decisions around Fabric, TurboModules, and native integration
- System design for mobile
- CI/CD, automation, distribution workflows
- Security best practices
- Communication, planning, conflict resolution

Common Interview Patterns in Product & Service Companies

Product Companies (FAANG, Unicorns, High-Scale Startups)

Expect deeper rounds like:

- JavaScript/TypeScript fundamentals & async challenges
- React internals, reconciliation, rendering strategies
- State management patterns, re-render optimization
- System design for mobile apps, scalability
- Architecture discussions (monorepo, modularization)
- Performance tuning: memory, FPS, bridging overhead
- Behavioral rounds

Service Companies (TCS, Accenture, Infosys, Wipro, etc.)

Focus is more practical and implementation-oriented:

- UI development, navigation, basic animations
- API integration, forms, validation, error handling
- Storage, offline sync, Redux basics
- Standard libraries and ecosystem knowledge
- Scenario-based problem-solving
- Faster interview cycles, fewer rounds

2. React Native Basics

Q1: What is React Native?

- React Native is an **open-source mobile application framework** used to build **Android and iOS apps**.
- It was **developed by Facebook (now Meta)** and first released in **2015**.
- React Native allows developers to write apps using **JavaScript or TypeScript**, following **React concepts** such as **components, props, state, and hooks**.
- It supports a **single codebase** that works on both **Android and iOS**, which **reduces development time and effort**.
- React Native uses **real native UI components** like **View, Text, and Image**, and **does not use HTML or WebView**, so apps look and feel like **native apps**.
- React Native applications provide **near-native performance**, which is suitable for most **business and consumer applications**.
- When required, developers can also write **native code** using **Kotlin/Java for Android** and **Swift/Objective-C for iOS**, giving more **flexibility**.
- React Native is **widely used in the industry** and is adopted by companies like **Facebook, Instagram, and Shopify**, with strong **community support**.

Q2: Why did you choose React Native for your projects?

1. Single codebase for multiple platforms

- React Native allows writing **one codebase**
- The same code works on:
  - Android
  - iOS
- This reduces development and maintenance effort

2. Faster development

- React Native speeds up development
- Features like **hot reloading** help see changes instantly
- Faster development leads to quicker releases

3. Near-native performance

- React Native uses **real native UI components**
- Performance is close to native apps
- Suitable for most real-world applications

4. Cost-effective solution

- One team can handle both platforms
- Less development time means lower cost
- Good choice for startups and businesses

5. Reusable and maintainable code

- Uses component-based architecture
- UI components can be reused
- Code is easier to manage and maintain

6. Strong community and ecosystem

- Large developer community
- Many third-party libraries available
- Easy to get support and solutions

7. Easy integration with native code

- Native code can be added when required
- Android → Kotlin / Java
- iOS → Swift / Objective-C
- Gives flexibility for platform-specific features

Q3: What is the difference between Native development and React Native?

Feature	Native Development	React Native
Platform	Android and iOS developed separately	Android and iOS from a single codebase
Programming Language	Kotlin/Java (Android), Swift/Objective-C (iOS)	JavaScript / TypeScript
Codebase	Separate codebase for each platform	Single shared codebase
UI Components	Direct use of platform-specific native UI	Uses real native UI components via React
Performance	Best possible native performance	Near-native performance
Development Speed	Slower due to separate development	Faster due to code sharing

3. JavaScript & TypeScript for React Native

Q:1 What are the different ways to declare variables in JavaScript?

In JavaScript, variables can be declared in **three main ways**:

1. `var`
2. `let`
3. `const`

1. var

- Introduced in **older versions of JavaScript**
- Has **function scope**
- Can be **re-declared and re-assigned**
- Gets **hoisted** (initialized as `undefined`)
- Not recommended in modern JavaScript

2. let

- Introduced in **ES6**
- Has **block scope**
- Can be **re-assigned**
- Cannot be **re-declared in the same scope**
- Safer than `var`

3. const

- Introduced in **ES6**
- Has **block scope**
- Used for values that should not change
- Cannot be **re-assigned**
- Must be **initialized at declaration**

Q:2 What is hoisting in JavaScript?

- **Hoisting** is a JavaScript behavior where **variable declarations are moved to the top of their scope** during the compilation phase.

- Only **declarations are hoisted**, not initializations.

How hoisting works

- JavaScript scans the code before execution
- Variable and function **declarations are registered first**
- Actual value assignment happens **at runtime**

Hoisting with `var`

- `var` declarations are hoisted and initialized with **`undefined`**

Hoisting with `let` and `const`

- `let` and `const` are **hoisted but not initialized**
- They stay in the **Temporal Dead Zone (TDZ)**
- Accessing them before declaration causes an error

Hoisting with `const`

- Same as `let`
- Additionally, `const` **must be initialized at declaration**

Real-World Analogy

- Hoisting is like **reserving seats before the event**
- The name is registered, but the person arrives later

Q3: What is block scope in JavaScript?

- **Block scope** means a variable is **accessible only inside the block** `{ }` where it is declared.

- Variables declared using **`let` and `const`** have block scope.

- A block can be:
  - `if` block
  - `for` / `while` loop
  - `{ }` block

Key Points

- `let` and `const` → **block-scoped**
- `var` → **not block-scoped** (function-scoped)
- Block scope helps prevent **unintended variable access and bugs**

Code Example:

Using `let` (Block Scoped)

```
if (true) {
  let x = 10;
}
console.log(x); // ✘ Error: x is not defined
```

Using `var` (Not Block Scoped)

```
if (true) {
  var y = 20;
}
console.log(y); // ✅ 20
```

Why block scope is important

- Prevents variable conflicts
- Makes code safer and predictable
- Improves readability and maintainability

Q:4 What is function scope in JavaScript?

- **Function scope** means a variable is **accessible only inside the function** where it is declared.
- Variables declared using **`var`** have function scope.
- `let` and `const` do **not** have function scope; they have block scope.

Key Points

- `var` → function-scoped
- Variables declared inside a function **cannot be accessed outside**
- Helps in **encapsulation of logic**

Code Example:

```
function test() {
  var x = 10;
  console.log(x); // ✅ 10
}
console.log(x); // ❌ Error: x is not defined
```

Why function scope matters

- Prevents variables from leaking outside functions
- Useful in older JavaScript code
- Can cause bugs when misused with loops

Q:5 What is global scope in JavaScript?

- **Global scope** means a variable is **accessible from anywhere in the program**.

4. React Fundamentals

Q1: What is the Virtual DOM in React?

- The **Virtual DOM** is a **lightweight copy of the real DOM**.
- It is a **JavaScript object representation** of the UI.
- React uses the Virtual DOM to **optimize UI updates** and improve performance.

Why Virtual DOM is needed

- Updating the **real DOM is slow**
- Frequent DOM updates reduce performance
- Virtual DOM minimizes **direct DOM manipulation**

How Virtual DOM works

1. React creates a **Virtual DOM tree** when the app loads
2. When state or props change:
    - A **new Virtual DOM** is created
3. React compares:
    - Old Virtual DOM vs New Virtual DOM
      (this process is called **diffing**)
4. React finds the **minimum changes**
5. Only those changes are applied to the **real DOM**
   (called **reconciliation**)

Key Points

- Virtual DOM is **faster than direct DOM updates**
- React updates only **changed components**
- Improves UI performance
- Works behind the scenes (developer doesn't manage it)

Q:2 What is the difference between Virtual DOM and Real DOM?

- **Real DOM** is the actual browser DOM.
- **Virtual DOM** is a lightweight JavaScript representation of the DOM.
- React uses the Virtual DOM to **optimize UI updates**.

Key Differences

Feature	Real DOM	Virtual DOM
Nature	Actual browser DOM	Lightweight JS object
Speed	Slow updates	Fast updates
Memory	More memory usage	Less memory usage
DOM manipulation	Direct and expensive	Batched and optimized
Re-rendering	Updates entire subtree	Updates only changed nodes
UI performance	Lower for frequent updates	Higher
Used by	Vanilla JS	React

How updates work

- **Real DOM**
  - Updates happen **immediately**
  - Each change triggers reflow and repaint

- **Virtual DOM**
  - Changes happen in memory first
  - React calculates minimal changes
  - Only necessary updates reach the real DOM

Q:3 What is reconciliation in React?

- **Reconciliation** is the process React uses to **update the UI efficiently**.

- When state or props change, React **compares the old Virtual DOM with the new Virtual DOM**.

- React then updates **only the changed parts** of the Real DOM.

Why reconciliation is needed

- Updating the Real DOM is **slow**
- Reconciliation finds the **minimum number of changes**
- Improves app performance and responsiveness

How reconciliation works

1. State or props change
2. React creates a **new Virtual DOM tree**
3. React compares:
    - Old Virtual DOM
    - New Virtual DOM
      (this is called **diffing**)
4. React identifies **what changed**
5. Only those changes are applied to the **Real DOM**

Key Points

- Reconciliation is based on the **Virtual DOM**
- Uses a **diffing algorithm**
- Updates are **batched and optimized**
- Happens automatically

Important Rules React Uses

- Elements of **different types** → whole subtree is replaced
- Elements of **same type** → only changed attributes are updated
- **Keys** help React identify list items efficiently

Q:4 What is the Fiber architecture in React?

- **Fiber** is the **new reconciliation engine** introduced in **React 16**.
- It allows React to **split rendering work into small units**.
- This makes React **faster, smoother, and more responsive**.

Why Fiber was introduced

- Old React used a **stack-based reconciliation** (blocking)
- Large UI updates could **freeze the UI**
- Fiber enables:
    - Incremental rendering
    - Better scheduling
    - Priority-based updates

What Fiber actually is

- Fiber is a **data structure**
- Each Fiber represents:
    - A component

5. UI Development in React Native

Q:1 What is the React Native rendering pipeline?

- The **React Native rendering pipeline** is the process by which **JavaScript code is converted into native UI** on Android and iOS.

- It defines **how a React Native component finally appears on the screen**.

Step-by-Step Rendering Pipeline

1. JavaScript Execution

- React Native code runs on the **JavaScript thread**.
- Components are written using:
    - React
    - JSX
- React creates a **Virtual Tree** of components.

👉 This step decides **what UI should look like**.

2. Shadow Tree Creation (Layout Phase)

- React Native creates a **Shadow Tree**.
- Shadow Tree:
    - Is not visible on screen
    - Used only for **layout calculations**
- Layout is calculated using **Yoga (Flexbox engine)**.

👉 This step decides **size and position** of UI elements.

3. Communication to Native (Old: Bridge / New: JSI)

- Layout and UI instructions are sent to native side:
    - Old architecture → **Bridge**
    - New architecture → **JSI (direct communication)**

👉 Data moves from JS to native efficiently.

4. Native UI Creation

- Native UI components are created:
    - Android → `View`, `TextView`
    - iOS → `UIView`, `UILabel`
- These are **real native components**, not WebView or HTML.

5. Rendering on Screen

- Native UI thread draws components on screen.
- User can interact with the UI (touch, scroll, gestures).

Q:2 What is the Bridge in React Native rendering?

- The **Bridge** is the communication layer that allows **JavaScript code to talk to native code** in React Native.

- It connects the **JavaScript thread** with **native modules and UI components**.

Why the Bridge was needed

- JavaScript and native code run in **different threads**
- They cannot directly talk to each other
- Bridge acts as a **messenger** between them

How the Bridge works

1. JavaScript creates UI updates or calls native APIs
2. Data is **serialized into JSON**
3. JSON messages are sent over the **Bridge**
4. Native side:
    - Deserializes the message
    - Executes native code
5. Results are sent back to JavaScript

Key Characteristics

- **Asynchronous** communication
- Uses **JSON serialization**
- One-way per message
- Shared by UI updates and native module calls

Limitations of the Bridge

- Serialization overhead
- Performance bottlenecks for heavy operations
- No direct synchronous calls
- Can cause UI lag for complex apps

Q:3 What is the View component in React Native?

- **View** is the **most basic and fundamental UI component** in React Native.
- It is used to **build layouts and containers**.

6. State Management

Q:1 What is the Local State in React Native?

- Local state is **data that belongs to a single component**
- It controls the **UI behavior and data** of that component only
- It is **not shared** across the entire app by default

Why Local State is Needed

- To handle **UI changes** like:
  - Input text
  - Button clicks
  - Toggle states
  - Loading indicators
- Without state, UI would be **static**
- Makes the app **interactive and dynamic**

How Local State Works

- When state changes:
  - React **re-renders** the component
  - UI updates automatically
- State is managed using:
  - `useState` hook (most common)
  - `this.state` in class components (older way)

Common Use Cases of Local State

- Form input values
- Show / hide password
- Loading spinner
- Modal open / close
- Checkbox or switch value

Common Mistakes

- Updating state directly
  ```
  count = count + 1
  ```
- Forgetting that state updates are async
- Using too much local state instead of lifting it up

Q:2 What is Redux?

- Redux is a **state management library**
- It is used to manage **global application state**
- Multiple components can **read and update the same data** from one place

Why Redux is Needed

- Passing props deeply becomes messy (**prop drilling**)
- Many screens need the **same data** (user info, cart, auth)
- State logic becomes hard to track in large apps
- Redux gives **predictable and centralized state**

Core Idea of Redux

- App has **one global store**
- State is **read-only**
- Changes happen in a **controlled way**

Three Core Principles

1. **Single Source of Truth**
    - Whole app state lives in **one store**
2. **State is Read-Only**
    - You cannot change state directly
    - You must send an **action**
3. **Changes via Pure Functions**
    - State changes happen through **reducers**
    - Reducers return a new state

Key Redux Components

1. Store

- Holds the entire app state

2. Action

- Plain JavaScript object
- Describes *what happened*

3. Reducer

- Function that decides *how state changes*

4. Dispatch

- Sends action to the store

How Redux Works (Flow)

1. UI triggers an action
2. Action is dispatched
3. Reducer receives action
4. Reducer returns new state
5. Store updates state
6. UI re-renders

When to Use Redux

- Large applications
- Shared data across many screens
- Complex state logic
- Authentication, cart, user profile

When NOT to Use Redux

- Small apps
- Simple UI state
- One or two screens only
- Local state is enough

Q:3 How do you create a Redux store?

- Store is the **central place** that holds the entire app state
- Only **one store** exists in a Redux application
- It connects **actions, reducers, and UI**

Basic Way to Create a Redux Store

1. Create a Reducer

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case "INCREMENT":
```

7. Data Persistence & Storage

Q: 1 What is AsyncStorage in React Native?

AsyncStorage is a **simple, unencrypted, asynchronous key–value storage system** in React Native.

It is used to **store small amounts of data locally on the device** and retrieve it later, even after the app is closed or restarted.

Why AsyncStorage is Needed

Mobile apps often need to store data that should:

- Persist after app restart
- Be available offline
- Be quickly accessible

AsyncStorage helps store such data **without a database setup**.

Common Use Cases

AsyncStorage is typically used for:

- Login tokens (JWT, auth token)
- User preferences (theme, language)
- App settings
- Onboarding status (first launch or not)
- Cached small API responses

⚠️ Not meant for large or sensitive data.

How AsyncStorage Works

- Data is stored as **key–value pairs**
- Both key and value are **strings**
- Operations are **asynchronous**
- Returns **Promises**

Internally:

- Android → uses SharedPreferences / SQLite
- iOS → uses native storage mechanisms

Limitations of AsyncStorage

- Not encrypted
- Not suitable for large data
- Slower than in-memory state
- No complex querying

👉 For secure data, use **Secure Storage / Keychain**

Q:2 Can you use async/await with AsyncStorage?

Yes. AsyncStorage methods return **Promises**, so they are designed to be used with **async/await**.

Why async/await Works with AsyncStorage

- All AsyncStorage APIs are **asynchronous**

- Methods like:
  - `setItem`
  - `getItem`
  - `removeItem`
  - `clear`
- Return a **Promise**
- `async/await` makes the code:
  - Easier to read
  - Easier to debug
  - Less nested than `.then()`

Q:3 What is SQLite?

SQLite is a **lightweight, embedded relational database** used to store structured data **locally on the device**.

It works **inside the app**, does not require a server, and stores data in a **single database file**.

Why SQLite is Used

Mobile apps often need to store:

- Large amounts of data
- Structured data with relationships
- Data that must be queried, filtered, sorted

AsyncStorage is not enough for this. SQLite solves that problem.

8. Networking in React Native

Q:1 What is the Fetch API in React Native?

The Fetch API in React Native is a **built-in JavaScript API** used to make **network requests** such as calling REST APIs, fetching data from a server, or sending data to a backend.

React Native uses the **same Fetch API standard as the browser**, so the syntax and behavior are very similar.

Key Characteristics of Fetch API

1. Built-In

- Fetch is **available by default** in React Native
- No installation required
- Works out of the box

2. Promise-Based

- Fetch returns a **Promise**
- Works naturally with:
    - `.then()`
    - `async / await`

```
fetch(url).then(response => response.json());
```

3. Asynchronous

- Network calls do **not block the UI**
- Runs in the background
- UI stays responsive

4. Platform Independent

- Same code works on:
    - Android
    - iOS
- React Native handles native networking internally

Q:2 What is Axios?

Axios is a **popular third-party JavaScript HTTP client library** used to make **network**

requests from applications, including **React Native** apps.

It is built on top of **Promises** and provides a **cleaner, more powerful API** compared to the native Fetch API.

Why Axios Is Used Instead of Fetch

While Fetch is built-in, Axios is often preferred because it:

- Reduces boilerplate
- Handles common use cases automatically
- Makes API handling more maintainable in large apps

Key Features of Axios

1. Promise-Based

- Supports `async / await`
- Cleaner error handling

2. Automatic JSON Transformation

- Automatically converts:
  - Request body → JSON
  - Response → JavaScript object

👉 No need to call `response.json()` manually.

3. HTTP Error Handling

- Automatically rejects the Promise for:
  - 4xx errors
  - 5xx errors

4. Interceptors

Interceptors allow you to:

- Modify requests before they are sent
- Handle responses globally
- Attach auth tokens
- Handle refresh tokens

9. Asynchronous Programming & Concurrency

Q:1 What is the JavaScript Event Loop?

The JavaScript Event Loop is the **mechanism that allows JavaScript to handle asynchronous operations** (like API calls, timers, promises) **without blocking the main thread**, even though JavaScript itself is **single-threaded**.

Why the Event Loop Is Needed

JavaScript is Single-Threaded

- JavaScript has **one call stack**
- It can execute **only one thing at a time**

If JavaScript waited for:

- Network requests
- Timers
- File reads

👉 The UI would freeze completely.

So the Event Loop exists to **keep the app responsive**.

Core Components of the Event Loop

To understand the Event Loop, you must understand **these parts together**:

1. Call Stack

- Where JavaScript executes code
- Functions are pushed and popped from here
- Runs **synchronous code**

2. Web APIs (Browser / RN Environment)

Provided by the environment, not JS itself.

Examples:

- `setTimeout`
- `fetch`

- DOM events
- WebSockets

These APIs:

- Handle async tasks
- Work in the background

3. Callback Queue (Task Queue / Macro-task Queue)

- Stores callbacks like:
  - `setTimeout`
  - `setInterval`
  - UI events

Callbacks wait here **until the call stack is empty**.

4. Microtask Queue

- Higher priority than callback queue
- Used by:
  - Promises (`then`, `catch`)
  - `queueMicrotask`
  - `MutationObserver`

👉 **Microtasks run before callbacks**

5. Event Loop (The Orchestrator)

The Event Loop:

1. Checks if the call stack is empty
2. Executes all microtasks
3. Executes one task from callback queue
4. Repeats forever

Q:2 What is the difference between macrotasks and microtasks in JavaScript?

👉 JavaScript always executes **microtasks before macrotasks** once the call stack becomes empty.

Macrotasks:

Macrotasks (also called the **task queue**) are **larger, scheduled tasks** that are executed **one at a time** by the Event Loop.

10. Architecture & Design Patterns

Q:1 What is MVVM architecture in React Native?

MVVM stands for **Model – View – ViewModel**.
It is an architectural pattern that separates **UI**, **business logic**, and **data handling** to make the app easier to build, test, and maintain.

Why is MVVM Used in React Native?

MVVM helps solve common problems in large React Native apps.

Problems without MVVM

- UI and logic mixed together
- Components become huge and hard to read
- Difficult testing
- Hard to reuse logic
- Bugs increase as app grows

MVVM solves this by

- Separating UI from business logic
- Making code modular and reusable
- Improving testability
- Making the app scalable

MVVM Components:

1. View – The UI Layer (What the user sees)

- The **View** is the visible part of the application.
- Responsible only for displaying data
- Handles user interaction (button clicks, input) and It is responsible for **displaying data**.

In React Native, the View is usually:

- A functional component
- Written in JSX
- Focused on layout, styles, and rendering state

What the View SHOULD do

- Render UI based on data it receives
- Trigger actions when the user interacts (button click, text input)

- Show loading, error, or success states

What the View SHOULD NOT do

- Make API calls directly
- Contain complex business rules
- Transform or validate data
- Handle side effects

2. ViewModel – The Brain of the Screen

The **ViewModel** is the most important part of MVVM.

- Acts as a bridge between View and Model
- Holds state and business logic
- Prepares data for UI

The ViewModel contains **state**, **business logic**, and **decision-making code**.

In React Native, a ViewModel is commonly implemented as:

- A custom hook
- A state container (Redux, Zustand, MobX)
- A combination of hooks and services

Responsibilities of ViewModel

- Fetch data from the Model
- Hold UI state (loading, error, data)
- Decide what data the View should see
- Expose functions that the View can call

Key idea

The ViewModel **does not know anything about UI layout**.
It only knows **what data exists** and **what actions are possible**.

Why this matters

Because logic is outside the UI:

- You can test it without rendering screens
- You can reuse it across multiple screens
- You can refactor UI without breaking logic

This is what makes MVVM feel **professional and scalable**.

3. Model – The Data and Business Rules Layer

The **Model** is responsible for everything related to data.

It does not care about:

- Screens
- Buttons
- User interactions

It only focuses on:

- Fetching data
- Database logic
- Updating data
- Applying business rules

In React Native, Model usually includes

- API services (Fetch / Axios)
- Database logic (SQLite, Realm)
- Data mappers
- Repository functions

Important concept

The Model **never talks directly to the View**.
It only communicates with the ViewModel.

How All Three Work Together

Let's understand this with a real scenario.

1. User opens a screen
2. View requests data by calling a function from ViewModel
3. ViewModel asks the Model to fetch data
4. Model fetches data from API or database
5. Model returns data to ViewModel
6. ViewModel updates its state
7. View automatically re-renders with new data

At no point does:

- The View call the API directly
- The Model update UI
- The ViewModel control layout

Each layer does **only its own job**.

Why This Separation Is Critical in Real Projects

In small apps, mixing everything inside one component may work.
In **real production apps**, this causes serious problems:

- Components become very large
- Logic duplication increases
- Bugs are hard to track
- New developers struggle to understand the code
- Testing becomes painful

MVVM prevents this by enforcing **discipline in code structure**.

Q2. What is the Difference Between MVC, MVP, and MVVM?

Aspect	MVC	MVP	MVVM
Full Form	Model – View – Controller	Model – View – Presenter	Model – View – ViewModel
Main Goal	Separate UI, data, and input handling	Improve separation and testability	Clean separation with reactive UI
Role of View	Displays UI and sends user actions	Passive UI, no logic	Displays UI and observes state
Role of Middle Layer	Controller handles user actions and updates Model	Presenter handles all logic and updates View	ViewModel manages state and business logic
Communication Style	View ↔ Controller ↔ Model	View → Presenter → Model	View ⇄ ViewModel → Model
View to Model Interaction	Often indirect but tightly coupled	No direct interaction	No direct interaction
Does middle layer update View directly?	Yes	Yes	No
How UI updates	Controller decides what View shows	Presenter explicitly updates View	View auto-updates when state changes
Coupling Level	High	Medium	Low

11. Package & Dependency Management

Q:1 What is package.json?

- `package.json` is the main configuration file used in Node.js and React Native projects.
- It stores important information about the project such as metadata, dependencies, scripts, and configuration.
- It acts as the “brain” of the project, helping tools and package managers understand how the project is structured and what it needs to run.

Why package.json Is Important

- Defines project name, version, and details
- Lists all libraries (dependencies) the project uses
- Helps install and manage packages using npm or yarn
- Stores scripts that automate tasks
- Ensures project setup consistency across systems
- Supports version control and collaboration

Role of package.json in React Native

- Manages React Native version
- Handles third-party libraries
- Tracks dependency versions
- Supports project scripts
- Ensures consistent environment setup

Q:2 What is the dependencies field?

- The `dependencies` field in `package.json` is a section where all the packages required for the application to run in production are listed.
- These libraries are installed automatically when someone runs `npm install` or `yarn install`.
- They are essential for the actual working of the app.

Why the dependencies Field Is Important

- It clearly defines which external libraries the app depends on
- Ensures consistent setup across all developers and environments

- Helps npm or yarn install the correct packages
- Keeps the project organized and maintainable

Example:

```
"dependencies": {
  "react": "18.2.0",
  "react-native": "0.73.0",
  "axios": "^1.6.0"
}
```

Key Points About dependencies

- Contains only packages needed at runtime
- These packages are bundled into the final build
- Version numbers define which release to install
- Updating dependencies may update app behavior
- Used in both React and React Native projects

Q:3 What are the main differences between npm, Yarn, and pnpm?

- npm, Yarn, and pnpm are JavaScript package managers.
- They are used to install, update, and manage dependencies in Node.js and React Native projects.
- All three do the same job but differ in speed, storage, and features.

npm (Node Package Manager)

- Default package manager that comes with Node.js
- Uses a `node_modules` folder to store packages
- Installs full copies of each dependency into every project
- Simple and widely supported
- Works well but may be slower with large projects

Strengths

- Pre-installed with Node.js
- Huge ecosystem
- Official standard tool

Yarn

- Created by Meta (Facebook) as an improvement over npm

12. Performance Optimization & Memory Management

Q:1 What is the Hermes engine?

- Hermes is a **lightweight JavaScript engine** developed by Meta specifically for React Native.

- It is optimized to make React Native apps **faster, smaller, and more memory-efficient**, especially on mobile devices.

- Hermes replaces the default JavaScript engine (like JavaScriptCore) to improve performance.

Why Hermes Was Created

- Mobile devices have limited CPU, RAM, and storage
- Traditional JS engines were optimized for browsers, not apps
- React Native needed a **mobile-first engine** focused on startup speed and performance

Key Features of Hermes

1. Ahead-Of-Time (AOT) Compilation

- JavaScript is compiled into bytecode at build time
- Reduces work needed at runtime
- Improves startup speed significantly

2. Smaller App Size

- Bytecode bundles are smaller than JS bundles
- Reduces APK/IPA size
- Saves storage space on devices

3. Faster Startup Time

- App loads quicker because parsing time is reduced
- Makes UI appear faster for users

4. Lower Memory Usage

- Hermes is designed to use less RAM
- Important for low-end Android devices

5. Better Debugging Support

- Works with React DevTools
- Includes its own debugger and profiling tools

How Hermes Works in React Native

- When enabled, React Native compiles JS into Hermes bytecode during build
- The engine loads bytecode instead of raw JS
- Less runtime parsing means better performance

When Hermes Is Most Useful

- Large apps
- Apps with many dependencies
- Apps targeting low-end devices
- Apps where startup time matters

Key Interview Points

- Hermes is a JavaScript engine built for React Native
- Improves startup speed and memory usage
- Uses ahead-of-time bytecode compilation
- Great for performance-critical apps
- Especially beneficial on Android devices

Q:2 What causes component re-renders in React?

A re-render happens when React decides that a component needs to update its UI.

This occurs whenever the data the component depends on **changes**.

Main Causes of Re-Renders

1. State Changes

- When `useState` value changes
- React re-renders that component
- And all of its child components (by default)

2. Props Changes

- If a parent passes new props to a child
- Even if the UI output doesn't change
- The child component re-renders

13. Testing in React Native

Q:1 What is unit testing in React Native?

- Unit testing is a testing method where you **test individual pieces of code (called units)** in isolation.

- In React Native, a “unit” usually means a function, component, hook, reducer, or utility module.

- The goal is to verify that each small part of the app behaves correctly on its own

What a Unit Test Checks

- Correct output for given inputs
- Component rendering behavior
- State and props logic
- Business logic correctness
- Edge-case handling
- Error handling paths

Why Unit Testing Is Important in React Native

- Catches bugs early during development
- Makes refactoring safer
- Improves code quality and reliability
- Prevents regressions when adding features
- Builds developer confidence
- Helps maintain large codebases

Example Areas Tested in React Native

- Pure functions (utilities, helpers)
- Reducers and selectors
- Hooks
- Components rendering states
- Validation logic
- Formatting functions

Tools Commonly Used

- Jest (most popular test runner)
- React Native Testing Library (for components)
- Enzyme (legacy in web/react)

Jest usually comes preconfigured in React Native projects.

Q:2 What is Jest and why is Jest popular for React Native testing?

- Jest is a **JavaScript testing framework** created and maintained by Meta.
- It is the **default and most widely used testing tool** for React and React Native applications.
- Jest runs tests, checks expectations, mocks modules, and reports results in a simple and efficient way.

Why Jest Is Popular for React Native Testing

1. Built-in Support for React Native

- React Native projects come preconfigured with Jest
- Minimal setup is required
- Works smoothly with React Native modules and components

2. Snapshot Testing Support

- Jest allows **snapshot testing**, which captures a rendered component's output
- Makes it easy to detect unintended UI changes
- Very useful for React components

3. Powerful Mocking System

- Can mock functions, APIs, timers, and modules
- Helps test components in isolation
- Removes dependency on real network calls or devices

4. Fast and Parallel Test Execution

- Runs tests in parallel
- Uses smart caching
- Ensures quick feedback during development

5. Simple and Developer-Friendly Syntax

- Easy to learn and read
- Works well with TypeScript
- Clear error messages and reports

6. Code Coverage Support

- Built-in code coverage reports
- Helps measure how much code is tested

14. Advanced React Native Topics

Q:1 What is a Native Module in React Native?

- A Native Module is a **bridge between JavaScript and platform-specific native code** in React Native.

- It allows you to write logic in **Java (Android), Kotlin (Android), Objective-C (iOS), or Swift (iOS)** and then call that logic from JavaScript.

- This is useful when React Native does not provide a built-in API for some device feature.

Why Native Modules Exist

- JavaScript cannot directly access device-level features
- Native platforms expose powerful APIs
- Native Modules allow React Native apps to use them

What Native Modules Are Commonly Used For

- Camera access
- Bluetooth
- Sensors (Gyroscope, Accelerometer)
- NFC
- Native UI components
- Push notifications
- File system and storage
- Media playback

How Native Modules Work

1. You write native code in Android or iOS
2. You expose native functions to React Native
3. JavaScript calls those functions like normal JS methods
4. The Bridge / JSI handles communication between JS and native code

When You Need Native Modules

- Platform-specific performance optimization
- Missing APIs in React Native
- Integration with third-party SDKs
- Hardware-level access

Q:2 What is the difference between Native Modules and JavaScript modules?

Native Modules and JavaScript modules both add functionality to a React Native app, but they run in **different environments** and are used for **different purposes**.

What Are JavaScript Modules?

- Written in JavaScript
- Run on the JavaScript thread
- Used for app logic, UI, state management, utilities, etc.
- Fully cross-platform (same code runs on Android and iOS)
- Do not directly access device hardware or OS APIs

Example:

```
import mathUtil from './mathUtil';
```

What Are Native Modules?

- Written in platform languages
    - Java / Kotlin → Android
    - Objective-C / Swift → iOS
- Run in the native layer of the OS
- Exposed to JavaScript via the Bridge or JSI
- Used to access device-level APIs or high-performance logic

Example:

```
NativeModules.DeviceInfo.getModel();
```

Q:3 What is a Native UI Component?

- A Native UI Component is a **custom user interface element created in native platform code** (Android or iOS) and then exposed for use inside a React Native app.

- It allows React Native developers to reuse or build UI elements that are not available in React Native by default, while still using them like normal React components.

What Makes It "Native"?

- It is implemented using
    - View / ViewGroup in Android
    - UIView in iOS

15. Application Security in React Native

Q:1 What is secure coding and Why is secure coding important in React Native?

- Secure coding is the practice of **writing code in a way that prevents security vulnerabilities and protects user data from attacks or misuse**.

- It means thinking about security while designing, developing, and testing your React Native app, not just after it is finished.

- The goal is to ensure the app remains safe even if attackers try to exploit weaknesses.

What Secure Coding Involves

- Validating all user inputs
- Protecting sensitive data
- Avoiding hard-coded credentials
- Encrypting communication
- Handling errors safely
- Following security best practices

It is a continuous mindset, not a single step.

Why Secure Coding Is Critical in React Native

1. Protects User Data

- Mobile apps often handle
    - personal information
    - passwords
    - financial data
- Secure coding ensures this data cannot be stolen or leaked

2. Prevents Hacking and Exploits

- Attackers target apps to
    - steal data
    - inject malicious code
    - tamper with APIs
- Secure code reduces vulnerabilities

3. Prevents Financial and Business Loss

- Security breaches cause

    - revenue loss
    - legal penalties
    - brand damage
- Prevention is far cheaper than recovery

Common Security Risks in React Native

- Storing secrets in AsyncStorage
- Unencrypted API communication
- Weak authentication flows
- JavaScript bundle reverse-engineering
- Insecure 3rd-party SDK usage
- Poor input validation
- Leaky logs and debug data

Why React Native Needs Extra Attention

- JavaScript bundle can be decompiled
- Mobile devices can be rooted or jailbroken
- Data may persist locally
- Many layers exist
    - JS
    - Native
    - API
    - Database

Security must be applied across all layers.

Q:2 What is SSL/TLS?

- SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are **security protocols that encrypt data sent between a client and a server over the internet**.

- TLS is the modern and more secure version of SSL, but people often use the term **SSL** to describe both.

- These protocols ensure that data cannot be read or modified while it is being transmitted.

What SSL/TLS Protects

- Login credentials
- Personal information
- Payment details
- API communication

16. React Native Ecosystem Knowledge

Q:1 What are the prerequisites for Google Play Store submission?

- Before you can publish a React Native (or any Android) app on the Google Play Store, there are a number of **technical, legal, and policy requirements** you must complete.

- These steps ensure your app is secure, high-quality, compliant, and ready for real users.

Developer Account Requirement

- You must create a **Google Play Console Developer Account**
- Requires a one-time registration fee
- Identity verification may be required
- Organization accounts are available for companies

Developer Account Requirement

- You must create a **Google Play Console Developer Account**
- Requires a one-time registration fee
- Identity verification may be required
- Organization accounts are available for companies

Package and Versioning Rules

- Unique applicationId (package name)
- Version code must be an integer and always increase
- Version name should match your release plan

Target and Compile SDK Requirements

- App must target **minimum API level required by Google**
- Must use recent Android SDK versions
- Apps should follow modern privacy and permission rules

Policy and Compliance Requirements

- Must comply with Google Play policies
- Content policy
- Security policy
- User data and privacy policy
- Ads policy (if using ads)
- Financial regulations (if processing payments)

Failure to comply may lead to rejection.

Privacy Policy Requirement

- Mandatory if your app
    - collects personal data
    - uses login
    - uses analytics
    - uses permissions like camera, location, storage
- The policy must be hosted online (URL required)

Data Safety Section

- You must declare
    - what data your app collects
    - how it is used
    - whether it is shared
    - whether it is encrypted
- This appears on your Play Store listing

Providing false information can lead to suspension.

App Content Classification

- Age rating questionnaire must be completed
- Content rating categories include
    - Everyone
    - Teen
    - Mature
    - Adult content policies must be respected

Store Listing Assets Required

- App name (title)
- Short description
- Full description
- App icon
- Feature graphic
- Screenshots
    - Phone screenshots mandatory
    - Tablet screenshots recommended
- Promo videos (optional)

Assets must follow quality guidelines.

17. DevOps for React Native

Q:1 What is Mobile DevOps?

- Mobile DevOps is the practice of combining **development, testing, deployment, monitoring, and operations workflows specifically for mobile applications**.

- It adapts DevOps principles to the unique challenges of Android and iOS development, helping teams deliver mobile apps faster, more reliably, and with better quality.

Why Mobile DevOps Exists

- Mobile apps depend on app stores
- Release cycles require approvals
- Devices and OS versions vary widely
- Native builds are complex
- Continuous updates and testing are needed

Mobile DevOps solves these challenges with automation and collaboration.

Key Areas Covered in Mobile DevOps

- Source code management
- Continuous Integration (CI)
- Continuous Delivery (CD)
- Automated testing
- Build automation
- Release management
- Monitoring and analytics
- Crash reporting
- Feedback loops

Typical Mobile DevOps Pipeline

1. Developer writes code
2. Code pushed to repository
3. CI system builds the app
4. Automated tests run
5. Signed release build is generated
6. Build is deployed to testers or stores
7. Analytics and crash logs are monitored
8. Feedback informs the next release

This cycle repeats continuously.

Why It Is Important for React Native Developers

- Cross-platform complexity
- Native dependencies
- Release signing
- OTA updates (CodePush etc.)
- Need for automated workflows

Mobile DevOps ensures consistency across platforms.

Q:2 What is the CI/CD pipeline in React Native?

- CI/CD stands for Continuous Integration and Continuous Delivery (or Deployment).

- In React Native, a CI/CD pipeline is an **automated workflow that builds, tests, and delivers your Android and iOS apps whenever code is updated**.

- The goal is to release high-quality mobile apps faster, with fewer manual steps and mistakes.

What Continuous Integration (CI) Means

- Developers push code to a shared repository
- An automated system
    - pulls the latest code
    - installs dependencies
    - runs tests
    - builds the app
- This ensures code works properly before merging

It keeps the project stable at all times.

What Continuous Delivery (CD) Means

- After CI succeeds
- The pipeline automatically
    - creates signed builds (APK/AAB/IPA)
    - distributes them to testers or stores
- This removes manual packaging work

Releases become predictable and repeatable.

18. Scenario-Based Interview Questions

Q1: How do you optimize a slow React Native app?

Scenario
Your app feels laggy while navigating screens or scrolling. The interviewer wants to know how you approach performance improvement step-by-step.

Answer

1. First, verify what is actually slow

I check whether the issue is:

- Slow navigation
- Slow list scrolling
- Delayed button clicks
- UI freezes during network calls

I use tools like Flipper or React DevTools to see:

- JS thread activity
- Re-render count
- Memory usage

So I am not guessing. I measure first.

2. Reduce unnecessary re-renders

Many times components update even when nothing important has changed.

I do things like:

- Use React.memo for list rows or UI-only components
- Use useCallback for functions passed as props
- Avoid keeping too much state at the top level

This keeps React from doing extra work.

3. Optimize lists

If the lag happens in lists, I:

- Use FlatList instead of ScrollView
- Provide keyExtractor with a stable id

- Use pagination instead of loading everything at once
- Avoid doing calculations inside renderItem

FlatList virtualizes rows so memory stays low.

4. Move heavy work away from UI

If I notice the JS thread is blocked, I check for:

- Large loops
- JSON parsing
- Image processing

Where possible, I:

- Move work to background
- Do processing before rendering
- Cache repeated results

This keeps the UI responsive.

5. Optimize images

I ensure:

- Images are compressed
- Caching is enabled
- Correct resolution is used

Large uncompressed images slow rendering.

6. Test in release mode

Debug builds run slower, so before final judgment I test in release.

Q2: How do you manage large API responses efficiently?

Scenario
You receive thousands of records in one API call and the app becomes slow.

Answer

1. Avoid loading everything at once

Instead of fetching the full list, I prefer:

- Pagination

- Infinite scroll
- Server-side filtering

This keeps memory stable.

2. Use FlatList virtualization

FlatList renders only visible items.
ScrollView renders everything.

So I always choose FlatList for large data.

3. Process data smartly

I avoid:

- Running heavy loops during render
- Re-parsing JSON multiple times

I:

- Clean and format data once
- Store processed data to reuse

4. Store large datasets efficiently

For persistent large data I use:

- SQLite
- Realm
- MMKV

AsyncStorage is fine only for small data.

5. Improve user experience

Instead of freezing, I:

- Show loading states
- Load pages gradually
- Avoid blocking UI

This keeps the app responsive and user-friendly.

Q3: How do you handle offline-first requirements?

Scenario
Your app must work even without the internet, and sync when connection returns.

Enjoyed This Preview?

If this sample helped you understand React Native concepts more clearly, you're going to love the full **React Native Interview Handbook**.

The complete book gives you:

✅ **500+ real interview questions**
✅ **Clear, simple explanations**
✅ **Beginner-friendly to advanced topics**
✅ **Practical, real-world scenarios**
✅ **Confidence to crack any Mobile Developer interview**

This preview is just the beginning.

The full version goes deeper, explains concepts step-by-step, and prepares you for real interviews so you don't just memorize answers, you actually understand them.

If you found value here, consider getting the complete book and unlock the full learning experience.

Your interview success journey starts here.

👉 **If you found this helpful, please purchase the full book to get complete access.**

If you enjoyed this, check out my other books

"Your Complete Guide to Cracking React Native Interviews"

About Author

Anand Gaur is a Mobile Tech Lead with rich experience in designing and developing impactful mobile applications. Skilled across Android, iOS, Flutter, and React Native, he has mentored many developers and guided them to crack interviews at leading IT companies.

You can find Anand at https://linktr.ee/anandgaur

ISBN : 978-93-5620-952-7

9 789356 209527