ULTIMATE

React Native for Cross Platform App Development

Turn Your Ideas into Scalable, Production-Ready Cross-Platform React Native Mobile Applications for iOS and Android

Felipe Becker Nunes

Copyright © 2026 Orange Education Pvt Ltd, AVA®

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor **Orange Education Pvt Ltd** or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Orange Education Pvt Ltd has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

First Published: April 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,

N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-46-6

ISBN (E-BOOK): 978-93-49887-37-4

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

1. Introduction and Environment

- Introduction
- Structure
- The Origin of React Native and Its Historical Context
- Expansion and Market Adoption
- Architecture
  - *New Architecture (JSI, TurboModules, and Fabric)*
- React Native 0.83.1 and React 19.2 Integration
    - *Other Notable Changes and Platform Updates in the 0.83.x Line*
- Advancing towards a More Dynamic Architecture
- Integration with Modern Technologies
- Monitoring and Debugging
- Enhancing React Native Development
- Use Cases
  - *Other Notable Examples*
- Benefits and Challenges of React Native
  - *Benefits of React Native*
  - *Challenges and Limitations of React Native*
- Expo CLI versus React Native CLI: Understanding the Two Approaches
    - *Expo CLI*
    - *React Native CLI*
  - *Approach to be Chosen*
    - *Chosen for this Book*
- JavaScript XML
  - *Basic Concept*
  - *Mixing Logic and Layout*
  - *Transformation into Function Calls*
  - *Advantages of the Declarative Style*
  - *Direct Use of Conditions and Loops*
  - *Best Practices*
- Setting up the Development Environment
- Creating Your First React Native Project
- Project Structure

- [Best Practices]
  - *Framework*
  - *Code Organization*
  - *Performance*
  - *Accessibility*
  - *Testing*
- Trends and Future
  - *Adoption of the New Architecture*
  - *Growth of the Expo Ecosystem*
  - *Cross-Platform Expansion beyond Mobile*
  - *Competition and Market Evolution*
  - *Impact on Mobile Development*
- Conclusion
- Self-Assessment Questions
- Key Terms

2. JavaScript/TypeScript and React Fundamentals

- Introduction
- Structure
- JavaScript Fundamentals (ES6+)
  - *The Road to ES6*
  - *Key Features Introduced by ES6*
  - *Variables and Scope (`Let`, `Const`, `Var`)*
  - *Arrow Functions and Their Use Cases*
  - *Template Literals*
  - *Destructuring (Arrays and Objects)*
  - *Spread and Rest Operators*
  - *Modules (Import/Export)*
    - *Exporting Modules*
    - *Importing Modules*
  - *Async/Await and Promises*
  - *JavaScript and TypeScript*
- Introduction to TypeScript
  - *Key Features and Benefits*
  - *Basic Types*
    - *String, Number, and Boolean*
    - *Array*

- *Tuple*
- *Enum*
- *Interfaces and Type Aliases*
  - *Interface*
  - *Type Alias*
- *Classes and Objects with TypeScript*
- *Thoughts about TypeScript*

Advanced TypeScript Features
- *Union Types*
- *Intersection Types*
- *Generics*
- *Generic Function*
- *Generic Interface*
- *Generics with Constraints*
- *Utility Types*
  - `Partial<T>`
  - `Readonly<T>`
  - `Pick<T, K>`
  - `Omit<T, K>`
- *Type Guards*
  - *Literal Types*
  - *What to Expect about TypeScript*

React Fundamentals
- *Historical Background of React*
- *Core Concepts of React*
- *Functional Components versus Class Components*
  - *Functional Components*
  - *Class Components*
- *Why Functional Components are Preferred*
- *Props: Passing Data between Components*
  - *Why Props Matter*
  - *Common Use Cases*
- *State: Managing Internal Component Data*
  - *State Management in Class Components*
  - *State in Functional Components*
- *The Importance of State*
- *Hooks and Component Lifecycle*

- *Managing State with useState*
  - *Controlling Side Effects with useEffect*
  - *Creating Custom Hooks*
  - *Hooks versus Class Lifecycle Methods*
  - *Thoughts on Hooks and Lifecycle*
- Conclusion
- Self-Assessment Questions
- Key Terms

3. Core React Native Concepts and Project Structure

- Introduction
- Structure
- How React Native Works Internally
  - *Native Modules and the Bridge Architecture*
  - *What is New in React Native 0.83.1*
- Best Practices for Code Organization in React Native Projects
  - *Component-Centered Architecture for Reusability*
  - *Separation of Concerns and Modular Organization*
  - *Animations and Performance with Reanimated*
  - *Internationalization (`i18n`)*
    - *Security*
    - *Other Key Points for Scalability*
    - *Reflections*
- Project Folder Structure
  - *Android-Specific Directories and Files*
  - *iOS-Specific Directories and Files*
    - *Shared Code Structure*
- Expo versus React Native CLI
  - *Pros and Cons of Expo*
  - *Ideal Scenarios for Using Expo*
  - *Steps to Migrate from Expo to Pure React Native CLI*
- Debugging Tools and Build Lifecycle
  - *Introduction to Debugging Methods*
    - *Fast Refresh*
  - *Overview of the Build Lifecycle and Deployment Strategies*
    - *Local Development*
    - *Testing and Validation*

Building the Application
Distribution and Release
Post-Launch Monitoring
Code Conventions and Project Standardization
Recommended Naming Conventions
Naming Recommendations
Global ESLint and Prettier Configuration
Maintaining Consistency in Large Teams and Projects
When Technical Mastery is not Enough: The Value of Organization
React Native as a Bridge: Between the Web and the Native World
Scalability is a Matter of Intention
Conclusion
Questions
Key Terms

4. Basic Components, Styling, and Responsive Layout
Introduction
Structure
React Native Basic Components
View Component: Structuring the Interface in React Native
The Role of View in Visual Organization
Common Issues and Bad Practices
Building a Complete Layout
Reflections on the View Component
Text Component: Displaying and Styling Text in React Native
Displaying Text Elements
Styling and Formatting Text
Specific Text Behaviors
Nesting and Advanced Formatting
Accessibility Considerations
Reflections on the Text Component
Image Component: Working with Images in React Native
Loading Local and Remote Images
Handling Image Assets Efficiently
Resizing with `resizeMode` 83
Image Optimization Checklist (React Native)

Reflections on the Image Component
TextInput Component: Capturing and Managing User Input
Customizing Inputs for Various Use Cases
Accessibility Considerations
Reflections about TextInput
`ScrollView` Component: Managing Scrollable Content in React Native
Handling Content Overflow with ScrollView
Nested Scrolling
Reflections about ScrollView
Styling in React Native
Using `StyleSheet 89`
Creating and Managing Styles
Best Practices for Organized Styles
Dynamic and Responsive Styling in React Native
Adapting to Different Screen Sizes
Using the Dimensions API
Advanced Responsiveness Techniques
Best Practices and Scenarios:
Final Thoughts
Conclusion
Self-Assessment Questions
Key Terms

5. Navigation with React Navigation
Introduction
Structure
Introduction to Navigation in React Native
Overview of React Navigation and Its Advantages
Initial Setup of React Navigation
Installing Core Packages
Additional Required Dependencies
Specific Configurations for Expo and React Native CLI
Wrapping the Application with NavigationContainer
Initial Setup Summary
Stack Navigation
Using `createNativeStackNavigator`

- *Comparison between `@react-navigation/native-stack` and `@react-navigation/stack`*
- *Customizing Screen Transitions and Animations*
  - *Advanced Customizations with `@react-navigation/stack`*
- *Choosing the Right Stack Navigator*
- *Stack Navigation Summary*

Tab Navigation
  - *Basic Implementation of `createBottomTabNavigator`*
- *Customizing Tab Icons*
- *Advanced Customization of Tab Styles*
- *Combining Navigators (Nested Navigation)*
- *Tab Navigation Summary*

Drawer Navigation
  - *Setting up `createDrawerNavigator` for Side Navigation*
- *Customizing Drawer Content with `drawerContent`*
  - *Styling and Positioning the Drawer*
  - *Combining Drawer Navigation with Stack Navigation*
  - *Why Combine Drawer and Stack*
  - *Best Practices When Combining Navigators*
- *Drawer Navigation Summary*

Passing Parameters between Screens
- *How to Send Parameters with `navigation.navigate`*
- *How to Access Parameters with `route.params`*
- *Using Hooks: `useNavigation` and `useRoute`*
  - *useNavigation*
  - *useRoute*
- *Best Practices When Using `useNavigation` and `useRoute`*
- *Why Hooks are Essential for Scalable Navigation Architectures*
  - *Important Tip: Optional Parameters*

Nested Navigation and Deep Linking
- *Setting up Nested Navigators*
  - *Example: Stack inside a Tab*
  - *Example: Tab inside a Drawer*

Handling Deep Links (Deep Linking)
- *Basic Deep Linking Configuration*

Customizing Headers and Transition Animations
- *Customizing Headers Using the `header` Property*

Dynamic Headers
Best Practices and Final Considerations
Structuring Navigation Efficiently for Scalability
Managing Navigation State and Preventing Undesired Behavior
Performance Optimization Tips for Complex Navigation
Best Practices Summary
Conclusion
Self-Assessment Questions
Key Terms

6. State Management with Redux and Context API
Introduction
Structure
Introduction to State Management
Why We Need State Management in Complex Apps
The Pitfalls of “Prop Drilling” and Data Inconsistencies
Overview: Redux versus Context API
Redux Fundamentals
Unidirectional Data Flow
Store: Concept and Responsibilities
Actions: Definition, Payloads, and Type Constants
Reducers: Pure Functions, Immutable State, and Composition
Composition of Reducers
Configuring Redux in React Native
Installing Dependencies
Creating the Store with `configureStore`
Wrapping Your App with `<Provider>`
Using the Primary Hooks: `useSelector` and `useDispatch`
Asynchronous Flow Management
HTTP Calls and Side Effects
Error Handling Strategies (Real-World Baseline)
Request Cancellation (`AbortController`)
Retry Logic for Transient Failures
Why Middleware Remains the Recommended Structure
Redux Thunk: Definition, Configuration, and Examples
Setup
Creating a `Thunk` Action

Usage in Component
Redux Saga: Generators, Watchers, and Workers
Setup
Quick Comparison: Thunk versus Saga
When to Use Redux Saga (And When Not To)
Redux Testing Practices
Testing Pure Reducers and Action Creators
Expanding Reducer Tests: Edge Cases, Parameterized Tests, and Snapshots
Snapshot Testing for Complex State Shapes
Mocking the Store for Component Tests
Testing Thunks and Sagas
Testing Thunks
Testing Sagas
Context API as a Lightweight Alternative
When Context Makes More Sense than Redux
Performance Caveat (Context)
Creating Contexts (`StateContext` and `DispatchContext`)
Custom Hook with `useContext`
Using `useReducer` Inside Context for Reducer-Style Logic
Integration of Context and Redux
Hybrid Uses (Context for Theming, Redux for Data)
Best Practices for Isolating Responsibilities
Best Practices for Global State Architecture
Selectors and Memoization (`reselect`)
Avoiding Unnecessary Re-renders (`React.memo`, `useCallback`)
Separation of UI Logic and Data Logic
File Naming and Organization
Final Considerations and Next Steps
Criteria for Deciding between Redux and Context in New Projects
Advanced Tools: Redux Toolkit and RTK Query
Conclusion
Questions
Key Terms

7. API Integration, Data Storage, and Offline Synchronization
Introduction

Structure
Introduction to API Integration
Difference between REST and GraphQL APIs
GraphQL
Typical Data Flow in Mobile Applications
Popular Libraries: Axios and Apollo Client
Axios
Apollo Client
Expanding API Versioning (and Rate Limiting) for Mobile Clients
Header-based Versioning (Media Type / Accept Header)
GraphQL "Versioning" (Schema Evolution Instead of Versioned Endpoints)
Version Sunset Policies and Client Migration Guidance
Handling API Rate Limiting in Mobile Apps
Common Signals
Recommended Client Behavior
Simple Retry Policy (Conceptual)
Best Practices and Advanced Considerations
Consuming REST APIs with Axios
Organizing Endpoints into Services
Handling Authentication, Errors, and Logging with Interceptors
Pagination, Filters, and Query Parameters
Consuming GraphQL APIs with Apollo
Core Concepts
Queries
Mutations
Subscriptions
Setting up Apollo Client
Installation
HTTP and WebSocket Links
ApolloProvider
Apollo Hooks
useQuery
useMutation
Cache Policies and Fetch Policies
Error Handling and Loading States
Patterns for Spinners and Progress Indicators

Displaying Error Messages to Users
Automatic Retry and Exponential Backoff
Failure Logging for Diagnostics
Local Data Storage
AsyncStorage: When to Use and Its Limitations
Full-Fledged Databases: SQLite and Realm
SQLite
Schema Definition
Setup and CRUD
Realm
Schema Definition
Basic CRUD Operations
Organizing the Data Access Layer
"Offline-First" Strategies
Principles of Offline-First
In-Memory Cache versus Persistent Cache
Responsive UI When Offline: Placeholders and Stale Data
Detecting Network Status with NetInfo
Data Synchronization on Reconnection
Pending Operations Queue (Write Queue)
Conflict Resolution Strategies
Manual Merge
Reconnection Events and Automatic Sync Triggers
User Feedback on Sync Progress
Testing and Best Practices
Unit Tests for API Services
Mocking Axios
Mocking Apollo Client
Integration Tests in an Offline Environment
Simulating Disconnection
Monitoring: Performance and Availability Metrics
Pre-Deployment Review Checklist
Conclusion
Self-Assessment Questions
Key Terms

8. Accessing Native Device Features

Introduction
Structure
Camera Access
- *Popular Libraries*
  - *React-Native-Camera*
  - *Expo-Camera*
- *Initial Setup*
  - *React Native CLI*
  - *Expo Managed Workflow*
- *Usage Example*
- *Advanced Customizations*

Geolocation
- *Recommended Libraries*
- *Permission and Manifest Configuration*
- *Practical Example*
- *Error Handling and Limitations*

Push Notifications
- *Local versus Remote Notifications*
- *Firebase Cloud Messaging*
- *Apple Push Notification Service*
- *Receiving and Displaying Notifications*
- *Deep Linking and Custom Actions*

Permission Management
- *Runtime Permissions*
- *Requesting and Revoking on iOS*
- *Supporting Libraries*
- *UX Best Practices for Permissions*

Security and Privacy
- *Handling Sensitive Data*
- *Responsible Use of Location and Media*
- *Privacy Policies and Regulatory Compliance*

Testing and Debugging
- *Testing on Simulator/Emulator versus Real Device*
- *Log Inspection Tools*
- *Mocking Native APIs*

Custom Native Modules and Bridging
Background Processing and Task Scheduling

Future Trends and Challenges
Edge Computing and On-Device AI
Privacy, Security, and Regulation
API Convergence and a Unified Experience
Performance, Energy Efficiency, and Offline UX
Conclusion
Self-Assessment Questions
Key Terms

9. Testing and Debugging
Introduction
Structure
Introduction to Testing and Debugging
Overview of the Test Pipeline
Tools and Workflows Used in This Chapter
Typical Workflow
Unit Tests with Jest
Initial Jest Configuration in React Native
Organization and Naming Conventions
Writing Basic Tests
Mocks and Spies `(jest.mock, jest.fn)`
Code Coverage
Integrating Jest into CI (GitHub Actions / CircleCI)
Integration and End-to-End Testing
What "Light E2E" Tests are and When to Use Them
Installing and Configuring the Integration Environment
Common Providers
"Emulating" Devices: Test Containers
Writing Your First Full-Flow Test
Synchronization and Waiting Strategies
Running Jest in CI
Best Practices and Performance Tips
Error Monitoring with Firebase Crashlytics
Overview of Crashlytics
SDK Integration in a React Native Project
Android Native Setup
Initialize in JavaScript

Capturing Unhandled Errors and Panics
Sending Custom Logs and Breadcrumbs
Analyzing Reports and Setting Alerts
Best Practices for Naming and Grouping
Real-Time Debugging
React Native Debugger
Element and Prop Inspection
Connecting
Redux and State Debugging
Enabling
Using
Breakpoints and Watch Expressions
Flipper
Database and `AsyncStorage` Inspection
Performance Monitor and CPU Profiler
Continuous Logging and Monitoring Strategies
Structured Logging Best Practices (JSON, Severity Levels)
Shipping Logs to External Services (Sentry, and Loggly)
Performance Metrics and Tracing
Uptime Dashboards and Alerts
Feedback Loop: From Production Error to Development Fix
Reflections and Future Trends
Recapping the Testing and Debugging Pipeline
Quality Checklist before Release
Evolving Your QA Strategy and Future Trends
Conclusion
Questions
Key Terms

10. Performance Optimization and Architecture Best Practices
Introduction
Structure
Introduction to Performance Optimization
Render Performance versus Network Performance
The Cost of Unnecessary Renders
Code Splitting and Lazy Loading
Code Splitting in React Native

Lazy Loading Heavy Components and Routes
Progressive Loading Strategies
List Optimization
Key Props for Performance
Boosting Performance with getItemLayout
Memoizing Components and Functions
Differences between Value and Function Memoization
`React.memo` for Functional Components
Comparative Example: With and Without Memoization
Code Organization and Folder Structure
Feature-Based versus Type-Based Organization
Naming Conventions and File Structure
Standardization with Linters and Formatters
Identifying Performance Bottlenecks
Analysis Tools: React DevTools, Flipper, Hermes Profiler
React DevTools
Flipper
Hermes Profiler
Using `why-did-you-render` and `Perf` Hooks
Architectural Patterns for Scalability
Context API versus Redux versus Zustand: When and Why
Best Practices for Layer Separation (View, State, Service, Data)
Performance Best Practices Checklist
Rules to Avoid Unnecessary Re-renders
Strategies for Animations, Images, and Intensive Events
Conclusion
Questions
Key Terms

11. Adding TypeScript and Advanced Integrations
Introduction
Structure
Introduction to TypeScript in React Native
Gradual Migration versus 100% TypeScript Projects
Benefits of Strong Typing for Maintenance and Scalability
Preparing the Project for TypeScript
Configuring `ESLint` and Prettier for TypeScript Projects

Adjustments to the Metro Bundler and Build Configuration Files
Incremental Migration from JavaScript to TypeScript
“File-by-File” versus “Whole-Project” Strategies
Handling Modules without Official Typings
Creating a 100% TypeScript React Native Project
Folder Structure and Naming Conventions
Best Practices for Organizing Code in TypeScript
Advanced Typing in React Native Components
Generic Components with `React.FC<… >` and Generics
TypeScript Utility Types (`Partial`, `Pick`, `Omit`, `Record`)
Higher-Order Components (HOCs) and Typing Higher-Order Functions
Data Modeling and Interfaces
API Response Types: DTOs and Schemas
Using Interface versus Type for Domain Entities
Integration with Validation Libraries (Zod, Yup) and Type Generation
Typings for External Libraries
Creating `.d.ts` Module Declarations for Untyped Libraries
Maintaining and Versioning Custom Typings
Advanced Integrations
QR Code Generation and Scanning (`react-native-qrcode-svg`)
Other Integrations: Camera, Sensors, and BLE with Typed Native Plugins
Managing React Native and Dependency Updates
Handling Breaking Changes and Typing Conflicts
Automating Regression Testing after Upgrades
Best Practices and Migration Patterns
Documenting Types: Using TSDoc and Generating Docstrings
Gradual Rollout Strategies and Production Error Monitoring
Conclusion
Self-Assessment Questions
Key Terms

12. Internationalization (i18n) and Accessibility
Introduction
Structure

Core Concepts of Internationalization
Setting Up `i18n` Libraries in React Native
- *Installing and Configuring i18n-js with TypeScript*
- *Organizing Translation JSON Files*
- *Using react-intl for More Advanced Needs*
- *Choosing between `i18n-js` and `react-intl`*

Translating Text and Components
- *Dynamic Language Switching*
- *Formatting Dates, Numbers, and Currencies*
- *Internationalized Date Formatting with `react-intl`*
- *Displaying Numbers across Cultures*
- *Proper Currency Display and Monetary Formatting*

Managing Internationalization State
- *Example: Persisting Language Preference with Context API and AsyncStorage*
- *Integrating `i18n` with Global State Managers (Zustand and Redux)*
  - *Example with Zustand*
  - *Example with Redux*

Introduction to Accessibility in Mobile Apps
- *Making Components Accessible*
- *Basic Accessibility: Making Components Readable by Screen Readers*
- *Describing Dynamic States with `accessibilityState`*
- *Improving Focus Navigation*
- *Testing with VoiceOver and TalkBack*

Inclusive Design in React Native Applications
- *Color and Contrast Accessibility*
- *Font Scaling and Readability*
- *Supporting Dark Mode*
- *High Contrast Mode Support*

Best Practices and Advanced Tips
- *Translation Management in Collaborative Environments*
- *Automating Translations with External Platforms*
- *Accessibility Audits and Continuous Testing*
- *Improving Complex Components*

Conclusion
Self-Assessment Questions

Key Terms

13. Build Automation and CI/CD
Introduction
Structure
CI/CD Fundamentals and Pipeline Structure
Branching Strategies: GitFlow versus Trunk-Based Development
End-to-End Flow: Build → Test → Deploy
GitHub Actions and Test Automation
Job and Step Structure
Test Integration
Coverage Reports and Status Badges
Fastlane: Automating Builds and Releases for iOS and Android
Defining Lanes for iOS and Android
iOS Lane: Build and Upload to TestFlight
Android Lane: Build and Upload to Google Play
Secure Certificate and Profile Management
Fully Automated Release in CI/CD Pipelines
EAS (Expo Application Services) para Build e Deploy
Integrating EAS Build with CI (GitHub Actions + Secrets)
Submitting Automatically with EAS Submit
Comparison: EAS CLI versus Fastlane (and How to Combine Them)
When to Combine EAS and Fastlane
Secure Key and Certificate Management
Generating a Keystore
Secure Versioning
Usage in CI (GitHub Actions)
Apple Certificates: Creation and Import in Fastlane / EAS
Creating a Certificate in Apple Developer Console
Importing Certificates in Fastlane with `match`
Importing into EAS
Secure Credential Storage
Security Best Practices and Key Rotation
Optimizations, Monitoring, and Next Steps
Job Parallelization and Artifact Cleanup
Build Notifications
Evolving Your Pipeline: Advanced Strategies

Conclusion
Questions
Key Terms

14. App Store Publishing and Maintenance
Introduction
Structure
Creating Developer Accounts
Apple Developer Program
Creating an Apple ID and Enrolling in the Program
Certificates and Provisioning for iOS
Google Play Developer Account
Setting up Your Google Play Developer Account
Managing Subscriptions and Payments
Preparing Your React Native App for Submission
App Icons and Assets
Specifications and Recommended Tools
Build Configuration
Configuration for Production (Release Builds)
Generating APK/AAB (Android) and IPA (iOS)
Using Tools such as Expo EAS or Fastlane
Submission and Review Process
Submitting to the App Store (iOS)
Configuring Your Application in App Store Connect
TestFlight for Beta Testing and Internal Distribution
Detailed Review Process of Apple
Submitting to Google Play (Android)
Internal, Closed, and Open Beta Testing with Google Play
Detailed Review Process of Google
Common App Store / Play Store pitfalls for React Native Apps
App Updates and Versioning
OTA Updates
Semantic Versioning Strategies
Submitting Incremental and Major Updates
Forced versus Optional Updates for Users
Best Practices for Release Notes in React Native
Analyzing App Usage and Metrics

Understanding Metrics in App Store Connect and Google Play Console
Downloads, Active Users, and Retention Rates
Crash Reports and Performance Monitoring
Integrating Analytics with React Native
Recommended Analytics Tools for React Native
Monitoring Specific User Behaviors and Events
Handling User Feedback
Collecting and Interpreting Reviews and Feedback
Store-Integrated Feedback Tools
Continuous Improvement from Real Feedback
Maintenance and Continuous Evolution Strategies
Bug Fixing and Codebase Maintenance
Adding New Features
Staying Compliant with Store Policies
Proactive Policy Change Management
Ensuring Continuous Compliance
Conclusion
Questions
Key Terms

15. Case Studies and Collaboration Best Practices
Introduction
Structure
From Theory to Practice
How Collaboration Impacts Project Success
Real-World Case Studies
Case 1: Cross-Platform E-commerce Application
Project Context
Key Architectural Decisions
Challenges and Solutions
Results (Example Outcomes; Measured Post-Launch)
Case 2: Health and Wellness App with Native Integrations
Project Context
Key Architectural Decisions
Challenges and Solutions
Results

Case 3: Interactive Education Platform
Project Context
Key Architectural Decisions
Mini-Case — The Cost of Speed: When Expo was Not Enough
Mini-Case — Performance That Pays Off: Early Optimization
Architectural Decisions and Trade-offs
Criteria for Choosing an Architecture
Monorepo versus Multiple Repositories
Modularization and Component Reuse
Performance Optimizations
Sacrificing Performance for Delivery Speed
Team Collaboration Best Practices
Version Control and Git Workflows
Code Reviews and Constructive Feedback
Code Standards and Quality Tools
Communication and Alignment
Continuous Integration and Continuous Delivery (CI/CD)
Risk Management and Contingency Planning in React Native Projects
Common Risk Categories
Risk Management Cycle
Example: Risk Scenario in Practice
Lessons Learned and Recommendations
Common Mistakes to Avoid
Strategies to Speed up New Member Onboarding
Practices to Ensure Long-Term Maintainability
Conclusion
Questions
Key Terms

16. Emerging Trends and the Future of React Native
Introduction
Structure
Evolution of the Ecosystem and New Libraries
The New Architecture Remains the Defining Platform Shift
Advanced Development Tools
Support Frameworks and Integration
Integration with Emerging Technologies

AR/VR in React Native (Current Landscape)
Development for Smartwatches and Wearable Devices
Technical Challenges and Solutions for Integration with Innovative Hardware
New Features and API Proposals
Proposals under Discussion and the Community Roadmap
Impacts on Performance, Interoperability, and Enterprise Adoption
Security, Privacy, and Compliance in React Native
Server-Driven UI, Superapps, and Multi-Experience Orchestration
Cross-Platform beyond Mobile: React Native for Desktop and Web
Cloud-Native Integration and Edge Computing
Observability, Monitoring, and Data-Driven Improvement
Integration of Artificial Intelligence with React Native
Technical Integration Strategies
Challenges, Ethics, and Best Practices
Future Directions
Community Contributions and Open Source
How to Collaborate with Libraries, Documentation, and RFCs
Examples of Successful Collective Contribution Initiatives
Sustainability and Longevity of React Native
Factors that Reinforce the Permanence of React Native in the Market
Adoption by Major Companies as an Indicator of Long-Term Viability
The Future of the Mobile Developer
The Profile of the Cross-Platform Developer in Evolving Market Scenarios
Preparing for Career Trends and New Opportunities
Conclusion
Questions
Key Terms

Index

CHAPTER 1

Introduction and Environment

Introduction

In this chapter, we will discuss the fundamental pillars of **React Native**, a **framework** that enables the development of mobile applications for different platforms from a single codebase. We will begin by revisiting the context in which React Native had emerged, and will proceed to exploring its history and evolution. Next, we will highlight how its internal architecture, based on a "**bridge**" that connects **JavaScript** to **native code**, has revolutionized the way mobile interfaces are created. We will also cover advantages, challenges, and real-world use cases, illustrating how companies from various industries have benefited from the framework. Finally, we will present best practices, integrations with other technologies, and future trends, providing a comprehensive overview of everything React Native offers for modern mobile development.

Structure

In this chapter, we will cover the following topics:

- The Origin of React Native and Its Historical Context
- Expansion and Market Adoption
- Architecture
- React Native 0.83.1 and React 19 Integration
- Advancing towards a More Dynamic Architecture
- Integration with Modern Technologies
- Monitoring and Debugging
- Enhancing React Native Development
- Use Cases
- Benefits and Challenges of React Native

- Expo CLI versus React Native CLI: Understanding the Two Approaches
- JavaScript XML
- Setting Up the Development Environment
- Creating Your First React Native Project
- Project Structure
- Best Practices
- Trends and Future

The Origin of React Native and Its Historical Context

The development of mobile applications gained significant momentum starting in the 2000s, becoming an essential part of our daily lives. Today, practically all aspects of life—education, entertainment, communication, banking services, healthcare, shopping, and transportation—can be mediated through mobile devices.

To keep up with this trend, the **official React Native documentation (https://reactnative.dev/)** and various studies on cross-platform applications report that different solutions emerged for app development, with React Native standing out as one of the most effective and influential options.

Originally released by **Facebook (now Meta)** in 2015, React Native aimed to solve one of the main challenges of mobile development: maintaining separate codes for Android and iOS. By leveraging the power of JavaScript and React, the technology significantly reduced redundant work, delivering applications that run on multiple platforms without sacrificing native features.

However, it is important to understand the landscape before React Native: in the mid-2000s, smartphones became widely adopted, with **iOS (Apple) and Android (Google)** emerging as the dominant mobile operating systems. Each ecosystem used its own language and tools—**Objective-C (later Swift) for iOS and Java (later Kotlin) for Android**—forcing companies that wanted to reach both platforms to duplicate efforts and maintain separate development teams.

To overcome these difficulties, hybrid solutions emerged, promising "*write once, run anywhere*" by using technologies such as **HTML, CSS, and JavaScript**. Tools such as **PhoneGap and Ionic** encapsulated web content within webviews, but they often failed to deliver the expected native performance, especially in animations and interactions.

According to accounts from the **Facebook development team**, the success of the React library on the web inspired the idea of applying it to mobile development. This led to the creation of a **communication bridge** that allowed JavaScript to interact with native system APIs, generating truly native components rather than just web pages disguised as apps.

It was in this context that **React Native was born**, introducing a **new way to build cross-platform applications**. Now that we have an understanding of the initial context, let us explore how React Native gained widespread adoption in the market.

Expansion and Market Adoption

After a period of internal development and a hackathon, Facebook officially introduced React Native to the public in 2015. The first demonstration took place in January of that year at the **React.js conference**, and by March, the framework was released as an **open-source project on GitHub**. Initially focused on **iOS**, React Native quickly expanded to support **Android** later that same year, allowing developers to build native applications for both platforms from a single JavaScript codebase.

From that point forward, major companies including **Instagram, Skype, Bloomberg, Tesla, and Pinterest** began integrating React Native into parts or even the entirety of their applications. The relatively smooth learning curve for JavaScript developers and the fast release cycle of new features contributed to the rise of the framework as a preferred choice for many development teams.

In 2016, there were plans to extend React Native beyond iOS and Android, with Microsoft collaborating to introduce support for **Windows** and Samsung exploring its use for **Tizen**. By 2017, the React Native community had grown significantly, despite occasional setbacks, such as temporary app store rejections—which, according to reports, were traced to third-party libraries rather than the framework itself. Nonetheless, the ecosystem

continued to expand, attracting thousands of contributors and a vast repository of packages and libraries.

Over the following years, the framework underwent continuous improvements. In 2018, the React Native team announced a **major architectural overhaul (New Architecture)** aimed at optimizing rendering and reducing communication bottlenecks between JavaScript and native code. This effort led to the creation of **Fabric** (a new rendering engine) and **Turbo Modules**, both designed to improve performance and responsiveness.

Meanwhile, in 2019, Facebook (now Meta) introduced **Hermes**, a JavaScript engine optimized for mobile devices, initially targeting **Android**. Hermes reduced startup time, memory usage, and even the binary size. Later, with support for **iOS (starting from React Native 0.64)**, reports indicated a 40% decrease in startup time on iPhones.

These initiatives demonstrate the ongoing commitment of Meta in improving React Native. Recent versions (0.70 and beyond) have adopted Hermes as the default JavaScript engine and integrated optional elements of the **new architecture**, ensuring that the framework remains up-to-date with the latest iOS and Android releases.

Today, React Native is firmly established as one of the leading mobile development tools, evolving with a focus on performance, developer experience, and platform versatility. With this foundation in mind, let us now delve into the architecture that underpins React Native and discover how it achieves native-like performance across multiple platforms.

Architecture

React Native renders real native UI on each platform (for example, **`UIView`** on iOS and **`ViewGroup`** on Android), rather than relying on webviews for the interface layer. JS–native communication has evolved from the legacy, batched asynchronous **Bridge** to the **New Architecture**, which removes the asynchronous Bridge and uses **JSI**, along with **TurboModules** and **Fabric** —often described as “bridgeless.”

This approach allows developers to achieve performance comparable to applications built in Swift/Objective-C or Kotlin/Java, without relying on web-views for the interface layer. In the current React Native release line

(0.83.x), the modern systems underpinning React Native are the **New Architecture** pillars—**JSI**, **TurboModules**, and **Fabric**—which replace the legacy Bridge-based model.

New Architecture (JSI, TurboModules, and Fabric)

- **JavaScript Interface (JSI):** The New Architecture removes the asynchronous **Bridge** and replaces it with **JSI**, which enables direct JS–native interaction via shared references and avoids Bridge serialization overhead; it can also support synchronous calls where appropriate.
- **TurboModules:** The native module system of the New Architecture, designed to modernize and streamline native-module integration and access patterns under the JSI-based model.
- **Fabric:** The New Architecture renderer that redesigns how native views are created and updated, improving rendering consistency and enabling closer alignment with modern React rendering capabilities.

Hermes as the Default JavaScript Engine

Hermes is used by default by React Native, and no additional configuration is required to enable it. For many apps, Hermes improves start-up time and reduces memory usage compared to JavaScriptCore; React Native also ships a Hermes build bundled to match each React Native release.

Improvements for iOS and Android

- **iOS Updates:** Recent versions of React Native improve compatibility with **iOS 15+** and support advanced system features (such as **Swift Concurrency**) while still maintaining compatibility with older versions.
- **Android Enhancements:** React Native now integrates better with **modern Gradle configurations** and Android libraries, reducing manual setup for projects using the CLI.

Interoperability with React

- **Concurrent Rendering:** The new architecture, along with Fabric, enables the use of the concurrent rendering features of React, thereby, improving UI responsiveness by allowing processes to pause and resume rendering when necessary, preventing noticeable blocking.
- **Suspense:** This feature simplifies asynchronous loading, allowing developers to display placeholders or loading states in a more declarative way.

Improvements in Developer Experience

- **Revised CLI:** The React Native team has refined the **official CLI**, making it easier to configure projects from scratch, especially for developers working with **native modules**.
- **Updated Documentation and Examples:** The official site receives frequent updates, providing clear guides on the **New Architecture**, along with step-by-step instructions for enabling **Fabric** and **TurboModules**.

Even as the platform evolves, the core ideas of React Native remain consistent:

1. **Native UI Rendering:** React Native renders real native UI on each platform (rather than running the interface in a webview).
2. **JS–Native Interoperability:** React Native coordinates JavaScript and native code so that UI and native capabilities can be updated in response to React state. *(Historically this used the "Bridge"; modern React Native emphasizes the New Architecture and a more direct, "bridgeless" model.)*
3. **Fast Refresh:** Code changes are reflected quickly on device or emulator to speed up UI iteration during development.

React Native 0.83.1 and React 19.2 Integration

At the time of writing, React Native **0.83** ships with **React 19.2**. In practice, this means you can use the React 19 feature set in mobile projects while also benefiting from the React 19.2 additions that React Native highlights for this release—most notably `<Activity>` and `useEffectEvent`. Key React 19 / 19.2 capabilities relevant to mobile are as follows:

- `Actions` **(Async Transitions):** React 19 adds support for using async functions in transitions ("`Actions`") to handle pending states, errors, and optimistic updates more ergonomically.
- `useActionState`**:** A hook for managing state derived from an Action result, including a built-in pending flag.
- `useOptimistic`**:** A hook for optimistic UI updates while an async action is in flight.
- `use`**:** A new API for reading promises or context during rendering; React can suspend while a promise resolves.
- `ref` **as a prop:** In React 19, function components can access `ref` as a regular prop, reducing reliance on `forwardRef`.

React 19 also introduces **React Compiler**, an optimization feature that enables **automatic memoization**, following the best practices of React for performance improvements. With **React Native 0.83**, React Compiler setup is simplified; developers only need to install the compiler and configure a **Babel plugin** in Metro, eliminating the need for multiple separate packages.

Other Notable Changes and Platform Updates in the 0.83.x Line

- **Developer tooling improvements (0.83):** React Native 0.83 introduces significant enhancements to React Native DevTools, including Network inspection and Performance tracing panels.
- **Web API support (0.83):** React Native 0.83 adds Web Performance APIs as stable and introduces `IntersectionObserver` support in Canary.

Advancing towards a More Dynamic Architecture

With React 19 integration, Hermes as the default engine, and the migration to JSI, TurboModules, and Fabric, React Native continues evolving toward a more dynamic and optimized architecture. The latest versions focus on:

- **Lower Complexity:** Shorter release cycles, fewer breaking changes, and easier adoption of new features.

- **Better Performance:** Support for **vector drawables**, improvements in rendering speed, and faster build times.
- **Enhanced Developer Experience:** Improved **debugging tools**, optional JavaScript logs, **React Compiler integration**, and better support for brownfield iOS apps.

Overall, with each new version, React Native reinforces its status as a powerful cross-platform development framework, combining high performance with native flexibility. For developers looking to migrate or start projects, **Upgrade Helper** provides detailed instructions for smoothly adopting new features across versions.

Having explored how React Native keeps evolving and improving, let us now examine the modern technologies and services that integrate seamlessly with it, enhancing both development efficiency and application capabilities.

Integration with Modern Technologies

The **React Native ecosystem** benefits from various tools and services that facilitate the development of robust mobile applications. Some of the most common integrations are as follows:

- **TypeScript:** The use of **TypeScript** has become almost a standard in many React Native projects, providing static typing and productivity enhancements such as auto-completion and safer refactoring. Adopting TypeScript is simple—developers only need to **add a `tsconfig.json` file** and use `.tsx` extensions for components. Many popular libraries already include TypeScript definitions, ensuring smooth integration. As a result, TypeScript helps prevent common runtime errors in JavaScript and promotes a clearer and a more reliable code.
- **GraphQL with Apollo or Relay:** React Native does not impose limitations on **API consumption**—both **REST** and **GraphQL** are supported. However, many developers prefer **GraphQL** due to its ability to fetch only the required data for each screen, reducing network overhead.

  Tools such as **Apollo Client** work in React Native the same way they do in web applications—by installing the client and configuring hooks

such as `useQuery` and `useMutation`. This setup even allows developers to share query logic across web and mobile apps. Large companies using React Native often adopt GraphQL with Apollo or Relay, leveraging a unified data management approach across multiple platforms.

- **Firebase:** For those looking for a scalable backend solution, **Firebase** offers authentication, real-time databases, storage, and notifications. The **React Native Firebase** project (community-maintained with official Google support) provides specific packages for each service, including **Analytics, Cloud Firestore, Cloud Functions, and Crashlytics**.

  With Firebase, developers can integrate backend functionalities without setting up their own servers. For example, installing `@react-native-firebase/auth` enables **authentication methods** (email, Google, Apple, and so on), leveraging **native SDKs** already integrated into Firebase.

  Additionally, Firebase provides push notifications via **Firebase Cloud Messaging (FCM)** for both Android and iOS, accelerating the development of advanced features.

Monitoring and Debugging

To effectively monitor, debug, and maintain a React Native app, developers typically combine production monitoring with modern local debugging tools:

- **Crashlytics and Sentry (production monitoring):** React Native apps commonly integrate Crashlytics (Firebase) and/or Sentry to capture JavaScript exceptions and native crashes, helping teams triage and fix issues before they affect more users.
- **React Native DevTools (current default debugger):** For local development, the recommended, built-in debugger of React Native is **React Native DevTools**. It replaces older debugging frontends such as the Flipper-based React Native plugin flow and prior debugger approaches, and provides integrated React DevTools capabilities plus dedicated panels for debugging.

- **Network and performance inspection (today):** In the 0.83 release line, React Native DevTools includes first-class **Network inspection** and **Performance tracing** panels. Network inspection captures calls from `fetch()`, `XMLHttpRequest`, and `<Image>`, and can show where a request originated; performance tracing provides a unified timeline view to diagnose runtime slowdowns.
- **Flipper (legacy / older codebases):** You may still encounter Flipper in older React Native projects or legacy setups. However, React Native removed native Flipper libraries/setup from **new projects** and the React Native Flipper plugin flow is **unsupported** for React Native-specific inspection features. Treat Flipper as a legacy tool you might need to understand when maintaining older apps, not the primary recommended debugger for modern projects.
- **Logging best practices:** Keep structured logs (locally and/or in a centralized sink) to investigate user-reported issues. Apply privacy and security best practices to avoid logging sensitive data.

Enhancing React Native Development

By integrating these technologies—whether for static typing, data consumption, backend integration, monitoring, or debugging—React Native becomes an even more comprehensive solution for modern mobile development.

Together, these tools create an ecosystem that supports:

- Faster development cycles.
- Greater reliability.
- Cloud and analytics support for production apps.

This enables a continuous cycle of development and improvement, making React Native a powerful choice for building scalable, high-performance mobile applications. Let us now explore how companies of various sizes—from startups to large corporations—are putting React Native into practice, illustrating its capabilities and versatility in real-world scenarios.

Use Cases

React Native, due to its maturity and popularity, has been adopted by companies of all sizes; from startups to large corporations, for large-scale applications. The following are some key examples that illustrate this adoption:

- **Meta (Facebook):** As the **original creator** of React Native, Meta integrates the framework into **strategic parts** of its ecosystem. It is used in products such as **Marketplace, Oculus VR applications, and Ad Managers**.

  Instagram, also owned by Meta, implemented React Native to launch features simultaneously on Android and iOS, demonstrating productivity gains and improved performance in specific modules.

- **Microsoft:** A major supporter of React Native, Microsoft maintains official versions for Windows and macOS and has migrated sections of its apps (such as **Office, Outlook, Teams, and Skype**) to React Native.

  This adoption allows Microsoft to synchronize updates across platforms while leveraging tools such as CodePush for **automated over-the-air (OTA) deployments**, accelerating the development cycle.

- **Tesla:** The mobile app of Tesla demonstrates how React Native can deliver near real-time responsive interfaces.

  The app provides remote control for vehicle functions (**locks, climate control**) and charging status, with simultaneous releases for iOS and Android. This approach reduced development efforts, while maintaining a consistent user experience.

Other Notable Examples

Many well-known apps rely on React Native, including:

- **Discord:** Voice and text communication.
- **Pinterest:** Image-sharing platform.
- **Uber Eats:** Restaurant management panel.
- **Shopify:** E-commerce platform.

React Native is also gaining traction in **finance (Coinbase, Bloomberg)** and is being integrated into high-demand services across various industries.

The versatility of React Native is evident in the wide range of companies that use it—from prototypes to complex applications.

Its ability to deliver cross-platform solutions quickly streamline product evolution that makes it a solid choice, even for large-scale implementations. Now, let us delve into the benefits and challenges of React Native. In the following section, we shall explore the key advantages that make React Native so appealing, as well as the challenges one needs to address when adopting this approach.

Benefits and Challenges of React Native

React Native bridges the productivity of web development with the performance of native applications. The technology allows developers to build apps for **both Android and iOS** using a shared JavaScript/React codebase while maintaining a high level of adherence to native interfaces. Let us now discuss the advantages that make React Native so appealing and the key challenges that may require attention when adopting this approach.

Benefits of React Native

React Native has become a popular choice for mobile development because it enables teams to build cross-platform applications with a shared codebase while still delivering a near-native user experience. It helps reduce development time, improve code reuse, and streamline maintenance across iOS and Android. These advantages make it especially valuable for projects that need faster delivery without sacrificing performance and usability.

Cross-Platform Development with a Single Codebase

- **Write once, run anywhere:** React Native enables developers to write screens and functionalities **only once**, significantly reducing the effort required to adapt the app for each platform.
- **Faster feature releases:** A single development team can build features for both Android and iOS, accelerating the release cycle.
- **Simplified maintenance:** Managing a single codebase reduces complexity compared to maintaining separate native apps.

Optimized Costs and Maintenance

- **Code reusability:** The same bug fixes and improvements apply to both platforms, reducing development costs and preventing inconsistencies between versions.
- **Time and resource savings:** Companies can launch products faster without compromising quality, making React Native a cost-effective solution.

Rapid Prototyping and MVP Development

- **Inspired by the web development model:** React Native allows developers to build **Minimum Viable Products (MVPs)** quickly, making it an excellent choice for startups and large enterprises looking to test market hypotheses before committing to full-scale app development.

Vibrant Ecosystem and Community Support

- **Extensive library support:** The popularity of React Native has led to a vast selection of open-source libraries and tools, making it easier to implement features such as maps, payments, and notifications.
- **Active community:** Developers benefit from a strong support network, including forums, groups, and extensive documentation.

Web Developer Productivity

- Developers familiar with React will find React Native easy to learn since it shares the same component-based and declarative state management approach.
- Web development tools such as VS Code, ESLint, Prettier, and package managers (npm/yarn) work similarly in React Native, ensuring a smooth workflow.

Seamless Integration with External Libraries

- **Firebase integration:** React Native works well with **Firebase** for authentication, real-time databases, and push notifications.
- **State management:** Redux and other state management solutions work without major complications.

- **Error tracking:** Services such as **Sentry and Crashlytics** help monitor and debug issues efficiently.

Challenges and Limitations of React Native

Despite its advantages, React Native also comes with trade-offs that teams should evaluate carefully before adoption. Some limitations arise from dependency on third-party libraries, differences in platform-specific behavior, and occasional need for native code integration. Understanding these challenges helps set realistic expectations and supports better architectural and planning decisions.

Performance Limitations for Graphically Intensive Applications

- **Almost-native performance:** React Native delivers near-native performance for everyday apps.
- **Limitations in 3D rendering and complex animations:** Apps that rely heavily on 3D graphics, physics calculations, or high-performance animations may experience delays due to the JavaScript-native bridge.
- Workarounds exist, but some projects opt for native development when high-performance graphics are essential.

Frequent Version Updates and Compatibility Maintenance

- **Rapid evolution of React Native:** The framework and its dependencies are frequently updated, requiring constant version upgrades to maintain compatibility with iOS and Android updates.
- **Potential for breaking changes:** Developers must regularly test and update dependencies to avoid compatibility issues.

Working with Native Modules

- **Accessing advanced native features (For Example, Cryptography, High-Performance Audio, Specialized Hardware):** These may require developers to write native code (Swift/Objective-C for iOS, Java/Kotlin for Android).
- **Additional expertise required:** Although React Native eliminates the need for separate native apps, teams may still need some native development knowledge.

App Size Considerations

- React Native apps tend to be slightly larger than fully native apps because they include JavaScript runtime and framework libraries.
- Optimization efforts are ongoing, but for projects where app size is a critical factor, this overhead may require consideration.

Library Fragmentation and Maintenance

- **Open-source dependencies:** While the ecosystem is rich, some third-party libraries may become outdated or lag behind React Native updates.
- Testing and manual fixes may be necessary to maintain stability.

Balancing Pros and Cons

Overall, React Native occupies an attractive middle ground:

- It delivers near-native performance for most use cases without sacrificing productivity.
- It outperforms hybrid solutions (such as webview-based apps) by rendering truly native components.
- It simplifies app maintenance and evolution, as both iOS and Android versions share a single codebase.

For business, consumer, or general-use applications, React Native delivers excellent results.

However, for projects requiring high-performance graphics, intensive native integrations, or extremely optimized applications, native development or alternative solutions may be more appropriate.

Even so, React Native remains a proven and powerful tool for:

- Reducing costs.
- Accelerating app launches.
- Maintaining a sustainable development workflow.

Building on these strengths, the ecosystem offers two primary approaches for initiating and managing projects: **Expo CLI** and **React Native CLI**. Let us explore each option in detail to help you decide which approach best meets your project requirements.

Expo CLI versus React Native CLI: Understanding the Two Approaches

The **React Native ecosystem** essentially offers two main approaches for starting and managing projects: **Expo CLI** and **React Native CLI**. This duality exists because, while some teams prioritize fast setup and abstraction of native configurations, others need full control over platform-specific details. **Expo CLI** provides a simplified experience—ideal for beginners or quick prototyping—whereas **React Native CLI** offers maximum flexibility for those who require advanced features and deep integrations with Android or iOS.

Expo CLI

- **Ease of Use and Speed:** Expo abstracts much of the Gradle (Android) and CocoaPods (iOS) configurations, speeding up project setup. It is particularly beneficial for new developers or teams looking to create prototypes without dealing with native code details.
- **Prebuilt Modules:** The Expo SDK includes built-in functionalities such as camera, location, notifications, and sensors. This allows developers to add features without installing extra libraries or manually configuring native code.
- **Build and Deployment:** Expo provides cloud-based services (EAS) for building Android/iOS apps, eliminating the need for local machines with native tools installed. It also supports Over-the-Air (OTA) updates, enabling bug fixes and feature updates without resubmitting the app to app stores.

About the limitations, some native modules may not be compatible or require **config plugins** when using the **bare workflow**. In such cases, developers can migrate to React Native CLI or use the bare workflow within Expo, ensuring greater customization flexibility.

React Native CLI

- **Full Flexibility:** Allows direct editing of native configuration files; Gradle for Android or `.plist`/`Info.plist` for iOS. This enables

detailed customizations and the addition of specific functionalities in brownfield applications, or when integrating SDKs and advanced hardware resources.

- **Deep Integration:** If a project requires advanced encryption libraries, proprietary APIs, or the latest native technologies, React Native CLI provides greater control. This level of customization is often essential for enterprise environments or solutions that demand high performance and security.
- **More Complex Setup:** Developers must manually install Android Studio (SDKs) and Xcode, along with additional configurations for each platform. Teams unfamiliar with native development may find the setup more time-consuming.

Table 1.1 shows the differences and main points in each approach:

Capability / Constraint	Expo Go	Expo Dev Build (Custom Dev Client)	Bare React Native CLI
Custom native modules?	No (limited to what is bundled in Expo Go + supported JS-only libraries)	Yes (add native code via config plugins and/or custom native projects)	Yes (full native freedom)
Third-party SDK constraints?	High (must be compatible with the runtime of Expo Go; many native SDKs will not work)	Medium/Low (most SDKs work; may require config plugins or native edits)	Low (direct integration of any SDK)
App Store builds?	Yes, but not using Expo Go itself; you build a standalone app	Yes (common/recommended for production in Expo workflows)	Yes (standard native build pipelines)
CI/CD?	Yes (commonly via EAS Build / EAS Submit)	Yes (commonly via EAS; also possible self-hosted)	Yes (Fastlane, GitHub Actions, Bitrise, CircleCI, and the like)
OTA updates?	Yes (via Expo Updates; subject to platform policy and architectural constraints)	Yes (same model as Expo Go—works with Expo Updates)	Optional (not built-in; you add something like CodePush or a custom solution)
Best for	Learning, quick prototyping, demos	Production apps needing native SDKs	Full native control, complex native requirements,

		while keeping Expo tooling	brownfield/embedded RN
Typical "switch" trigger	You need a native SDK / custom native module	You need deeper native customization than config plugins allow	You are already fully native, or requirements demand direct native control

Table 1.1: *Main Points in Each Approach*

In practice, "Expo" can mean two different local-development experiences: **Expo Go** (the standard shared client) and **Expo Development Builds** (a custom dev client compiled with your native dependencies). Teams typically start with Expo Go for speed, move to a Dev Build when they need native SDKs/modules, and choose Bare React Native CLI when they require full native control or are integrating into the existing native apps.

Approach to be Chosen

- **For Learning or Prototyping:** Expo CLI is generally better suited for beginners or those who need to validate an MVP quickly. The ease of starting and lack of complex native configurations lower the entry barrier.
- **For Large-Scale Apps / Advanced Native Features:** When an app requires features not directly supported by Expo (such as proprietary native modules or specific build configurations), **React Native CLI** offers full control over native code.
- **Gradual Migration:** It is possible to start with Expo for its fast setup and, if limitations arise, migrate to the bare workflow or React Native CLI. Most JavaScript code remains reusable, ensuring a smooth transition.

Chosen for this Book

Expo CLI is the fastest and simplest way to create React Native apps without worrying about native configurations; while **React Native CLI** is ideal for developers who need deep platform control.

Both options are officially supported, and the choice depends on whether the team wants to simplify the experience or requires advanced customization capabilities. However, in this book, we will be using **Expo**

CLI because it is easier to understand and offers a quicker learning curve. For more details, you can consult https://expo.dev/.

JavaScript XML

Before diving into your first project, it is essential to understand that JavaScript XML (JSX) is a syntax extension that allows you to write UI components in a way that resembles HTML while still leveraging the full power of JavaScript. Let us see about it in the next section.

Another important point before creating our first project is understanding that JavaScript XML (JSX) is a syntax extension adopted by React (and consequently by React Native) that resembles HTML but internally converts each element into JavaScript function calls. Despite its friendly appearance, JSX is not pure HTML—it acts as a declarative layer that simplifies the creation of complex interfaces, allowing the use of variables, conditional expressions, loops, and other JavaScript features directly within the "layout" of components.

Basic Concept

- Instead of `<div>` and `<span>` (as in the web), React Native uses `<View>` and `<Text>` to represent containers and text. JSX recognizes these elements and converts them into actual native components when executed.
- Unlike HTML, where attributes follow specific rules, JSX of React Native uses camelCase for properties and callback functions (`onPress`, `onLayout`, and so on), reflecting JavaScript conventions.

Mixing Logic and Layout

JSX allows JavaScript logic to be inserted directly into the interface structure. This means you can use conditional structures, loops, functions, and other constructs without needing to concatenate HTML strings, as was common in older front-end approaches. For example:

```
import React from 'react';
import { Text } from 'react-native';
export default function Hello() {
```

```
const name = 'Alice';
 return (
<Text>Hello, {name}!</Text>
 );
}
```

In this example, JSX inserts the value of the local variable `name` directly into the `<Text>` component, displaying the message **"Hello, Alice!"** without relying on any props or external variables. Thus, the interpolation syntax `{name}` allows React Native to render **Alice** declaratively, keeping the code simple and easy to understand.

Transformation into Function Calls

- When JSX code is processed, each snippet such as `<View>` or `<Text>` is transformed into an equivalent function call, such as `React.createElement(View, {… props}, … )`. In the case of React Native, these calls do not generate HTML nodes but rather structures that the framework interprets to build and update native components.
- This automatic conversion is typically handled by **Babel** (in the default configuration of React Native), eliminating the need to manually create HTML or use DOM manipulation functions, as was common in other libraries.

Advantages of the Declarative Style

- **Simplified Reading:** Combining layout and logic in a single file makes it easier to understand how the interface is built and updated.
- **Composition Flexibility:** With JSX, creating smaller and reusable components becomes natural, as each component can be imported as `<MyComponent />` and embedded within another.
- **Reduced Maintenance:** The complexity of handling HTML/markup strings and manually updating each element is replaced by a **declarative model**—simply change the state or props, and JSX reflects the updates in the interface.

Direct Use of Conditions and Loops

One of the significant advantages of JSX is that it allows the use of JavaScript features to control what is displayed, when, and how many elements are rendered. For example:

```
import { View, Text } from 'react-native';

export default function ListNames() {
constnomes = ['Alice', 'Bob', 'Charlie'];
  return (
    <View>
      {nomes.map((nome, index) => (
       <Text key={index}>Olá, {nome}!</Text>
      ))}
    </View>
  );
}
```

In this example, we have a component that displays a fixed list of names, where each name is transformed into a **`<Text>`** element with a unique identifier **`(key={index})`** to prevent React warnings and optimize rendering.

Best Practices

- **Componentization:** It divides the layout into smaller, more specific components (for instance, separate a list into **`<ListaNomes>`** and an individual item **`<ItemNome>`**).
- **Rendering Optimizations:** For large lists or performance-intensive components, using hooks such as **`React.memo`**, **`useCallback`**, or **`useMemo`** helps prevent unnecessary re-renders, ensuring optimal performance.
- **Consistent Styling:** Maintains a standardized naming convention (such as **`container`**, **`title`**, and so on) and use **`StyleSheet.create(…)`** to organize and reuse styles, keeping JSX cleaner.

JSX is fundamental in providing a development experience similar to the web ecosystem while focusing on creating native interfaces. It eliminates

the need to manipulate HTML nodes directly and simplifies the integration of logic (conditionals, loops, variables) with the layout structure. Combined with the declarative model of React, JSX enables developers to build React Native applications in a more readable, scalable, and efficient way, aligning mobile interface development with modern web application practices.

We will read about **components** in this section. Components are the building blocks of any React Native application. They are self-contained, reusable pieces of code that define how a particular section of your user interface should appear and behave. Think of them as individual “widgets” or “modules” that, when combined, create a complete, dynamic application.

At this point, you do not need to worry about mastering every detail of components. Throughout the book, we will progressively explore how to create, compose, and optimize components. Our goal is to gradually introduce you to these concepts so that you become comfortable with the idea of breaking down your interface into manageable, reusable parts.

For now, just remember that components help organize your code and make it more maintainable by isolating functionality and appearance. As we move forward, you will gain hands-on experience with components and learn best practices for structuring and integrating them, ensuring that your applications are both robust and scalable. Let us create our first project in React Native now!

Setting up the Development Environment

Properly configuring the development environment is the first step to ensuring that your React Native project runs smoothly throughout the coding, testing, and debugging phases.

- **Hardware Requirements**: To emulate Android or iOS, it is recommended to have a computer with at least **8GB of RAM** (ideally **16GB**) and a modern processor. Simulating mobile devices is resource-intensive, requiring significant CPU and memory usage.
- **Operating System**: You can develop on **Windows, macOS, or Linux**, but each system has specific requirements:
    - To compile **iOS**, you need a **Mac** with **Xcode** installed.

    - On **Windows and Linux**, iOS development is not officially supported.
- **Node.js**: Node.js is essential for running JavaScript outside the browser. Install the **LTS version** from https://nodejs.org and verify the installation with the commands on the terminal: `node -v` and `npm -v`.
- **Expo CLI**: Responsible for simplifying the creation of React Native projects, eliminating the need for manual Gradle (Android) and pods (iOS) configurations.
- **Code Editor**: **Visual Studio Code (VS Code)** is widely used due to its helpful extensions (ESLint, Prettier, Git) and native **TypeScript support**.
- **Emulators and Simulators**:
    - **Android**: Requires **Android Studio**, where you configure the SDK and create an **Android Virtual Device (AVD)**.
    - **iOS**: For **macOS users**, **Xcode** provides a built-in iOS simulator ready to use.

Now that you have familiarized yourself with the simulators required for testing on both Android and iOS, it is time to put your environment into action. In the next section, we will guide you through creating your first React Native project, laying the groundwork for your journey into mobile app development.

Creating Your First React Native Project

For beginners, the simplest way to get hands-on experience with React Native is by using **Expo**, as this platform abstracts various environment configurations. The following is a basic guide:

1. **Install Node.js**
    a. Visit nodejs.org and download the Long-Term Support (LTS) version.
    b. After installation, verify in the terminal with **`node -v`** and **`npm -v`** to ensure everything is set up.
2. **Expo CLI**

   a. Currently, instead of installing **`expo-cli`** globally, the recommended approach is to use **`create-expo-app`**, which comes with Expo updates.

3. **Create the project**

   a. In the terminal, run:

   ```
   npx create-expo-app MyProject --template blank
   ```

   b. It may ask about installing the latest version of the **`create-expo-app`** package. Confirm by pressing **y**.

   c. This will create a new project using the default **blank** template in JavaScript. If you want to use a different template, run:

   ```
   npx create-expo-app --template
   ```

   and replace **`--template`** with the desired one.

   d. This command generates the basic app structure, including folders like **assets** (for images and fonts) and files such as **App.js**.

4. **Navigate to the folder**

   a. Use **`cd MyProject`** to access the newly created project directory.

5. **Run the project**

   a. Use **`npm start`** or **`npx expo start`** to launch **Metro Bundler**, a web interface that manages your compilation of the app.

   b. Metro provides instructions to open emulators or scan a QR code with **Expo Go**.

6. **Test on Physical Devices or Emulators**

   a. **Physical Device:** Install the **Expo Go** app from the Play Store (Android) or App Store (iOS). Ensure your device is on the same Wi-Fi network as your computer, then point the camera at the QR code displayed in Metro.

   b. **Emulator:** If you prefer running on an **Android emulator** or **iOS simulator**, set them up (**Android Studio / Xcode**) and click "**`Run on Android device/emulator`**" or "**`Run on iOS simulator`**" in the Metro interface.

Once you have successfully run your application on an emulator or simulator, it is time to take a closer look at the underlying project structure. In the next section, we will walk through the typical folder organization created by Expo (as shown in *Figure 1.1*), which will help you understand where each piece of your project resides and how to navigate and customize it for your development needs.

Project Structure

After creating your project with Expo, a typical folder organization includes the following folders, we can see that in *Figure 1.1*.

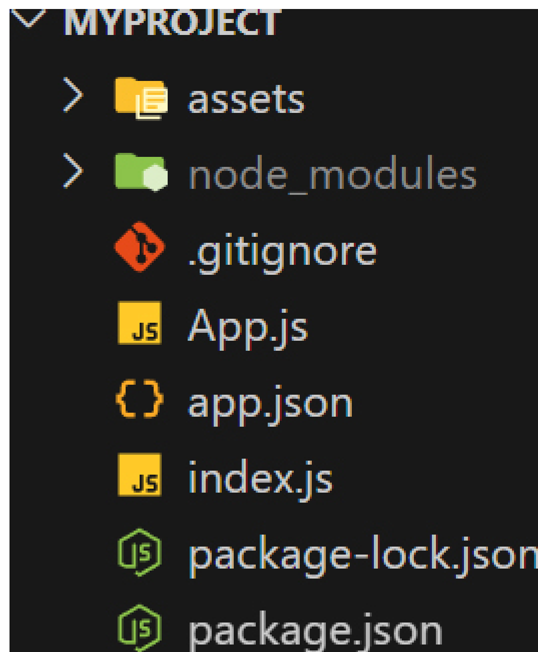

Figure 1.1: *Basic Structure of a New Project on React Native*

- **`.git/`:** It is a hidden folder that stores the entire Git versioning history and metadata. It is created when initializing a repository **`git init`** or cloning a project.
- **`assets/`:** It is a folder dedicated to static resources such as images, icons, fonts, and other files used in the application. Expo/React Native projects usually contain images (logos, illustrations) and custom fonts.
- **`node_modules/`:** It is the directory where dependencies installed via **`npm`** (or **`yarn`**) are stored. It contains all the libraries and packages required for the project, along with their subdependencies.

- **`.gitignore`:** This is a text file specifying which files or folders Git should ignore (not version). It typically includes `node_modules/`, log files, build artifacts and other sensitive or transient information.
- **`App` (JavaScript file):** It is the main application file (`App.js` or `App.jsx`). It usually serves as the entry point, defining the root component of the project while importing React Native libraries and other modules.
- **`app` (file or folder):** This may be another JavaScript file (say `app.js`) or a folder containing subcomponents. The name suggests it stores core logic or additional components. Its exact function varies depending on the conventions of the development team.
- **`index` (JavaScript file) :** Generally, it is an initial file that registers the main `App` component. In some cases, it registers it with `AppRegistry` (in pure React Native) or initializes specific services. It may act as a bridge between native configuration and the main component.
- **`package` (JSON file):** Typically named `package.json`, this file describes the name, version, execution scripts, dependencies, and project metadata of the app. If only named **"package"**, the `.json` extension might be hidden or renamed. Ensure it is actually `package.json`.
- **`package-lock` (JSON file):** Usually named `package-lock.json`. It records the exact versions of installed dependencies and subdependencies in `npm`, ensuring reproducible installations across different machines.

In summary, this structure reflects the basic components of a JavaScript project: version control (Git), resource folders (assets), dependencies (node_modules), main application files (App, index), and configuration files (.gitignore, package.json/package-lock.json). As your app grows, it becomes natural to add directories to keep the code modular. Let us talk about the best practices, now that we have already created our first project.

Best Practices

To achieve high-quality results with React Native, it is crucial to adopt best practices that cover everything from project organization to monitoring the application in production. Before diving into each aspect, rest assured that we will cover all of these best practices in detail throughout the book. There is no need to memorize or fully grasp them immediately; we will gradually explore how to implement and integrate these principles into your React Native workflow. In the following subsection, there are some recommendations.

Framework

- **Componentization:** Screens, cards, buttons, and lists can be divided into smaller, reusable components.
- **Separation of Responsibilities:** Business logic and access to external services should be placed in service modules, not inside interface components.
- **Document Routes:** If using React Navigation, keep screen definitions organized to facilitate future maintenance.
- **Consistent Styling:** Use `StyleSheet` or libraries such as Styled Components to standardize layout and styling.
- **Version Control:** Always use Git for project versioning.

Code Organization

- **Folder Structure:** Separating components, services, screens, and modules coherently (like `components`, `screens`, and `services`) improves scalability.
- **TypeScript:** Using static typing prevents errors and speeds up development.
- **Hooks and Functional Components:** Prefer functional components and hooks over class components to better leverage the features of React and keep the code concise.
- **State Management:** For complex apps, adopting solutions such as Redux, MobX, or Context API helps keep state logic organized.

Performance

- **Optimized Lists:** Use `FlatList` and `SectionList` for large collections instead of `ScrollView` with extensive data.
- **Conditional Rendering:** Minimize unnecessary re-renders using `React.memo`, `PureComponent`, or `shouldComponentUpdate` in critical areas.
- **Native Animations:** Use `Animated` with `useNativeDriver: true` or libraries such as React Native Reanimated for smooth transitions and gestures.
- **Image Caching:** Implement image caching (for example, with `react-native-fast-image`) and optimized formats (WebP) to reduce download size and memory usage.

Accessibility

- **Screen Readers:** Add properties such as `accessible={true}`, `accessibilityLabel`, and `accessibilityRole` to interactive or informational components.
- **Inclusive Layout:** Ensure proper contrast, font resizing, and clear icon descriptions.
- **Verification Tools:** Use Accessibility Inspector (iOS) or Accessibility Scanner (Android) to identify and fix navigation flow issues.

Testing

- **Multiple Levels:** Cover business logic with unit tests using Jest and the UI with libraries such as React Native Testing Library.
- **End-to-End (E2E):** Automate end-to-end flows with tools such as Detox or Appium, simulating real interactions on devices or emulators.
- **Continuous Integration:** Integrate tests into the CI pipeline to detect regressions before release.

By applying these practices, you can build robust, high-performance, and sustainable React Native applications, maximizing the benefits of the technology while minimizing the risks associated with cross-platform development. With a strong foundation in place, let us now look ahead.

React Native continues to evolve, meeting the ever-changing demands of the mobile market and unlocking new possibilities for developers. In the next section, we will explore the key trends and future perspectives that are shaping the ongoing development of the framework.

Trends and Future

React Native continues to evolve steadily, meeting the demands of the mobile market and offering new possibilities for developers. The main trends and future perspectives for the framework are explained in the following subsection.

Adoption of the New Architecture

React Native is transitioning to a redesigned architecture, which includes **JSI, TurboModules, and Fabric**, aiming to improve performance and interoperability. By reducing dependency on the asynchronous bridge between JavaScript and native code, the framework is expected to achieve even lower latency in UI operations, enabling smoother and more synchronized animations, even in demanding scenarios.

The new **Fabric renderer** also supports modern React features such as **Concurrent Rendering** and **Suspense**, facilitating the development of even more responsive applications. Although full adoption is gradual—since it depends on adjustments to existing libraries and native code—it is expected that, in the coming years, most apps will run in **`"bridgeless"` mode**.

Meta (formerly Facebook) continues to invest in improvements for this transition, adopting **Hermes** as the default JavaScript engine and optimizing modules to become **TurboModules**. It is important to note that the classic architecture remains supported, ensuring backward compatibility for those who prefer a more gradual migration.

Growth of the Expo Ecosystem

Expo has gained traction as a key platform within the React Native ecosystem, simplifying development at various levels. It provides an **SDK** packed with ready-to-use modules like camera, sensors, authentication, cloud-based build services, and **over-the-air (OTA) updates**.

With the release of features such as **Development Builds** and **Config Plugins**, Expo has become useful not only for rapid prototyping but also for large-scale enterprise applications. Recent studies indicate that over 70% of React Native developers use Expo at some stage of their workflow, whether for quickly starting projects or managing CI/CD pipelines with **Expo Application Services (EAS)**.

The distinction between **"bare RN"** and **"Expo-based"** is becoming increasingly blurred, positioning Expo as the preferred solution for starting —and often evolving—React Native applications.

Cross-Platform Expansion beyond Mobile

The "learn once, use anywhere" concept is materializing as React Native extends beyond **Android** and **iOS**. Official support for Windows and macOS has been introduced, thanks to the initiative of Microsoft. This allows developers to share business logic between desktop and mobile versions of the same app, as seen in **Microsoft Teams**.

At the same time, projects such as **React Native Web** enable RN components to run in **browsers**, while **Apple TV** and **Android TV** benefit from community-driven adaptations. There are even cases of React Native being used in **VR glasses** and **IoT devices**.

The idea of centralizing development in React/JavaScript while selectively adapting for different platforms is appealing to companies seeking consistency across multiple platforms.

Competition and Market Evolution

The cross-platform mobile development segment is highly competitive, with solutions such as **Flutter (Google), Jetpack Compose Multiplatform (Kotlin), and .NET MAUI (Microsoft)**. Each brings a different approach, but this competition drives React Native to innovate continuously, closing performance gaps and improving the developer experience.

For example, the growth of Flutter has pushed React Native to accelerate initiatives such as the **Fabric renderer** and startup optimizations. Meanwhile, the huge installed base of RN and its reliance on **JavaScript**

(the most popular language of the world) ensures that many companies have no incentive to migrate to another platform.

Interestingly, there is a convergence of ideas:

- Features pioneered by React Native (such as **hot reload**) are now seen in competitors.
- React Native, in turn, incorporates innovations from other frameworks.

With a strong community and backing from tech giants such as **Meta and Microsoft**, React Native is expected to remain a leading cross-platform framework.

Impact on Mobile Development

React Native has permanently changed mobile app development by integrating web development practices (like fast updates, high iteration levels) into the native ecosystem. This approach has enabled more teams to adopt cross-platform solutions, reducing costs and delivery time.

App stores have adapted their update policies, facilitating fast bug fixes (such as via **CodePush**). Looking ahead, as React Native becomes more robust—whether through its new architecture or tools such as Expo—its market presence is expected to grow.

The ability to unify web and mobile development in JavaScript is extremely attractive from an organizational perspective, driving adoption even in large-scale products.

React Native maintains a strong position thanks to continuous technical improvements and a thriving ecosystem. **Meta** continuously enhances the foundation of the framework, while the community expands its functionalities and best practices.

For developers and companies investing in React Native today, the future looks promising. Far from stagnating, RN is expanding its capabilities in performance and platform reach, solidifying itself as a reliable choice for building cross-platform mobile experiences.

Conclusion

In this chapter, we followed React Native from its origins, when Facebook faced the challenge of unifying code for iOS and Android, to its widespread adoption by companies of various sizes and industries. The idea of combining JavaScript with native components not only increased team productivity—previously hindered by duplicated efforts—but also preserved the performance and smoothness expected in a top-tier app.

Throughout the reading, we have seen that React Native has made significant progress, achieving better integration with platforms (Android and iOS), along with tools such as **Hermes, JSI, Fabric, and TurboModules**, which make mobile development even faster and more responsive. Features such as **Hot Reload** and **Fast Refresh** have optimized the testing and interface refinement cycle, while the large community ensures a rich supply of libraries, guides, and support.

We also explored real-world use cases, support for modern technologies (**TypeScript, GraphQL, Firebase, and so on**), and the benefits and limitations of the framework. We found that in scenarios requiring extreme graphical performance or extensive use of highly specific hardware resources, native development may still be preferable. However, for most applications, React Native strikes a balance between high performance, cost reduction, and faster feature deployment.

Finally, we analyzed best practices for code organization, testing, accessibility, and CI/CD, as well as the dual approach of **Expo CLI versus React Native CLI**—on one hand, the convenience of starting projects without dealing with native configurations; on the other, the flexibility to customize every aspect of the build. Each decision depends on the context and goals of the project.

With this foundation, it is clear that React Native is a robust and continuously evolving tool. It is a highly viable alternative for developing cross-platform applications, combining the JavaScript/React ecosystem with native components to ensure a smooth user experience. This versatility, coupled with a strong community and continuous improvements from Meta and contributors, reinforces React Native as one of the top choices for building modern and scalable mobile applications.

Before diving into React Native, it is essential to have a solid foundation in JavaScript and TypeScript. The next chapter reviews key ES6+ concepts such as arrow functions, async/await, destructuring, and modules. We will

introduce TypeScript, and demonstrate the advantages of typing your code in medium to large projects. The second part of the chapter will cover React fundamentals, including functional components, props, state, and the component lifecycle. This knowledge will be critical for building robust, maintainable applications in React Native.

Self-Assessment Questions

Before consolidating what has been studied throughout this chapter, it is useful to reflect on how React Native handles mobile app development, its differences from other approaches, and the tools that optimize the development workflow. The following questions serve as a guide to review the fundamental concepts presented, identify any doubts, and reinforce the understanding of the structure and best practices within the React Native ecosystem:

1. In what ways does React Native differ from hybrid applications based on WebView?
2. What is the importance of Hot Reload in the development cycle?
3. What factors make the React Native CLI more suitable than the Expo CLI in certain cases?
4. What are the main elements that make up the structure of a React Native project?
5. What are the main advantages of working with React Native compared to other hybrid or native solutions?

Answering these questions helps review and consolidate the material, allowing you to assess whether you have satisfactorily understood the concepts covered or if you need to explore any specific aspect further.

Key Terms

- **JavaScript XML (JSX):** A syntax extension that allows writing UI components using a structure similar to HTML, which is later converted into JavaScript function calls.
- **Hot Reload/Fast Refresh:** Features that enable instant code updates without losing the application state, improving development speed and

debugging efficiency.

- **Hermes:** A lightweight and optimized JavaScript engine designed specifically for React Native to improve startup time and execution performance.
- **JavaScript Interface (JSI):** A new abstraction layer that enhances communication between JavaScript and native code, replacing the old bridge architecture.
- **Fabric:** A redesigned rendering engine in React Native that enables smoother animations, lower latency, and better integration with modern React features such as Concurrent Rendering and Suspense.
- **TurboModules:** A new way of handling native modules, improving performance by making JavaScript-native interactions more efficient.
- **Expo CLI:** A tool that simplifies React Native development by handling native configurations, allowing quick project setup without requiring direct interaction with native code.
- **React Native CLI:** The standard command-line interface for React Native, offering greater flexibility and control over native configurations.
- **Flexbox:** A layout system used in React Native (similar to CSS Flexbox) that helps create responsive and adaptable UI structures.
- **React Navigation:** A popular library for managing navigation in React Native apps, offering support for stacks, tabs, and drawer navigation.
- **State Management (Redux, Context API, MobX):** Approaches used to manage application state efficiently, improving data flow and UI consistency.
- **Expo Application Services (EAS):** A cloud-based service that simplifies building, updating, and deploying React Native apps without requiring direct access to native build tools.
- **CodePush:** A technology that enables over-the-air (OTA) updates for React Native apps, allowing bug fixes and minor changes to be deployed without requiring a full app store update.
- **End-to-End Testing (E2E):** Testing methodology using tools such as Detox or Appium to simulate real user interactions in mobile

applications.

- **React Native Web:** A project that allows React Native components to be used in web applications, promoting code reuse across mobile and browser platforms.
- **Platform-Specific Code:** The ability to write code that runs only on specific platforms (Android/iOS) using platform checks or file extensions such as `.ios.js` and `.android.js`.
- **Over-the-Air (OTA) Updates:** A method of updating apps dynamically without requiring users to download a new version from the app store.

These key terms summarize some of the essential concepts covered in this chapter, helping reinforce the understanding of the ecosystem and development practices of React Native.

You’ve Just Finished your Free Sample

Enjoyed the preview?

Buy: http://www.ebooks2go.com