HANDBOOK

Modern App Development Approaches

Native, cross-platform, hybrid, PWA and web architectures explained with case studies and architecture diagrams.

Architecture Patterns · Case Studies · Architecture Diagrams · Decision Framework

5
Approaches

4
Architecture Patterns

3
Case Studies

Curated by Angela Guilherme

September 2026 Edition · Revised October 2026

Abstract. This handbook explains the five principal approaches to building modern applications (native, cross-platform, hybrid, progressive web apps and web applications), together with the internal architecture patterns that apply across them. Each approach is presented with its definition, technology stack, trade-offs and a layered architecture figure; a decision framework and three real-world case studies show how the approaches are matched to requirements. This revision, prepared in October 2026, updates framework versions and status (React Native, Flutter, Kotlin Multiplatform, Compose Multiplatform, .NET MAUI), the Apple and Android platform tooling, web rendering practice, and the role of AI-assisted development.

Contents

1. Introduction to Modern Application Architectures 1
2. Internal Architecture Patterns 2
3. Native App Development 3
4. Cross-Platform App Development 4
5. Hybrid App Development 5
6. Progressive Web Apps (PWAs) 6
7. Web Applications 7
8. Architecture Decision Framework 8
9. Real-World Case Studies 8
10. Industry Trends and Future Directions 9
Appendix A. Glossary of Key Terms 10
References and Sources 11

1. Introduction to Modern Application Architectures

1.1 What is application architecture?

Application architecture is the structural design of a software system: the arrangement of its components, how they interact, and the principles that govern their design. In mobile and web contexts it determines how the user interface, business logic, data access and platform APIs are organised and integrated. The choice of architecture is among the most consequential decisions in a project, because it affects performance, cost, time to market, team structure, maintainability and the end-user experience.

Principle. There is no universally best architecture. Each approach is a different set of trade-offs, optimised for particular use cases, team capabilities and business constraints.

1.2 The evolution of app development

The history of application development is a continuing tension between performance, portability and developer productivity. Early mobile applications were exclusively native. As the web matured, developers sought to reuse web technologies, producing hybrid apps and later progressive web apps. Newer approaches have been added alongside the older ones rather than replacing them, so the choice today depends on context (Table 1).

Table 1. *Evolution of application development approaches. Eras overlap and dates are approximate.*

Era	Dominant approach	Key technologies	Primary driver
2007–2012	Native mobile	Objective-C, Java	Launch of the iPhone (2007), the App Store (2008) and Android
2010–2015	Hybrid apps	Cordova, PhoneGap, Ionic	Adoption by web developers
2015–2018	Cross-platform frameworks	React Native, Xamarin	Code sharing across platforms
2018–2021	Flutter and PWAs	Flutter, Dart, Service Workers	Performance and reach

Era	Dominant approach	Key technologies	Primary driver
2021–2025	Declarative UI and Kotlin sharing	SwiftUI, Jetpack Compose, Kotlin Multiplatform	Modern native toolkits; shared logic with native UI
2025–present	Multi-approach with AI tooling	All of the above, Compose Multiplatform, AI coding agents	Context-specific choice; faster iteration

1.3 The five approaches at a glance

Table 2. *Summary of the five approaches covered in this handbook.*

Approach	Built with	How it runs	Distribution
Native	Swift (iOS); Kotlin (Android); Objective-C and Java in legacy code	Compiled for one platform, calling its SDK directly	App stores
Cross-platform	Dart (Flutter), JavaScript/TypeScript (React Native), C# (.NET MAUI), Kotlin (Kotlin Multiplatform)	One codebase, rendered by the framework or mapped to native UI on each platform	App stores
Hybrid	HTML, CSS and JavaScript with Capacitor, Cordova or Ionic	Web UI inside a native WebView shell	App stores
PWA	Standard web technologies, Service Worker, Web App Manifest	Installable web app in the browser engine	The web, optionally installed to the home screen
Web app	HTML, CSS, JavaScript; SPA or MPA frameworks	In the browser, no installation	A URL

2. Internal Architecture Patterns

Beyond the platform approach, every application needs an internal pattern that governs how the interface, business logic and data layer are separated and how they communicate. These patterns apply to native, cross-platform and hybrid apps alike (Table 3).

Table 3. *Architecture patterns compared.*

Pattern	How data flows	Commonly associated with
MVC	The Controller mediates between View and Model	iOS UIKit
MVP	The View talks to a Presenter through an interface, which talks to the Model	Android before Jetpack
MVVM	The View binds to observable ViewModel state	SwiftUI, Jetpack Compose
MVI	Intent produces new state, which the View renders (one direction)	Kotlin with Coroutines and Flow

2.1 MVC: Model-View-Controller

The foundational pattern. The Model holds data and business rules, the View renders the interface, and the Controller mediates between them. UIKit is built around MVC. It is simple, widely understood and separates concerns clearly, but the Controller can grow into a “Massive View Controller” and is tightly coupled to the View.

2.2 MVP: Model-View-Presenter

The Presenter takes over the Controller’s role but references the View only through an interface. This makes the Presenter highly testable, since it has no platform dependencies, at the cost of more boilerplate and many interfaces. MVP was popular on Android before Jetpack.

2.3 MVVM: Model-View-ViewModel

The ViewModel exposes observable state to which the View binds reactively. It has excellent testability and reduces boilerplate, and is the natural partner of SwiftUI and Jetpack Compose. Data binding can be hard to debug, and the pattern can be excessive for simple screens.

2.4 MVI: Model-View-Intent

MVI uses unidirectional data flow: user Intents are processed into a new state (the Model), which the View renders. State is predictable and debuggable, actions are easy to replay in tests and the pattern scales to complex interfaces, but it has a steeper learning curve and considerable boilerplate for simple screens. It is popular in Kotlin apps using Coroutines and Flow.

3. Native App Development

3.1 Definition and core concepts

A native application is built for a single operating system using that platform's official languages, SDKs and design guidelines. It is compiled for the target platform and calls its APIs directly, with no cross-platform abstraction layer. iOS apps use Swift (or legacy Objective-C), Xcode and the iOS SDK; Android apps use Kotlin (or Java), Android Studio and the Android SDK. Each platform has its own UI paradigms, UIKit and SwiftUI on iOS, and Jetpack Compose and the View system on Android, which must be learned separately.

3.2 Platform status in 2026

Apple announced iOS 27 and Xcode 27 at WWDC in June 2026, with further SwiftUI additions such as new document APIs, resizable sheets and a revised @State implementation [4]. SwiftUI is now the default for new iOS interfaces, although UIKit remains in use for complex or legacy screens, and Swift Concurrency (async/await) has largely superseded callback-based code. On Android, Kotlin is the primary language and Jetpack Compose is the standard UI toolkit, with Android Gradle Plugin 9 and Kotlin 2.x as current build requirements [2]. On-device machine-learning frameworks (Core ML and Apple's on-device foundation models on iOS, Gemini Nano and ML Kit on Android) increasingly make local AI a native advantage.

Table 4. *iOS development stack.*

Layer	Technologies	Purpose
UI framework	SwiftUI, UIKit	Interface construction and navigation
Language	Swift, Objective-C	Application logic
Concurrency	Swift Concurrency (async/await), Combine	Asynchronous work and reactive data
Persistence	SwiftData, Core Data, SQLite	Local storage and sync
Networking	URLSession, Alamofire	HTTP, REST and GraphQL
Media, AR, ML	AVFoundation, ARKit, Vision, Core ML	Capture, playback, augmented reality, on-device ML
Distribution	Xcode and App Store Connect	Build, sign and distribute

Table 5. *Android development stack.*

Layer	Technologies	Purpose
UI framework	Jetpack Compose, XML Views	Declarative and imperative UI
Language	Kotlin, Java	Application logic; Kotlin adds null safety
Concurrency	Coroutines, Flow	Asynchronous programming and reactive streams
Persistence	Room, DataStore, SQLite	Type-safe local data
Networking	Retrofit, OkHttp, Ktor	REST and GraphQL clients
Media	Media3 (ExoPlayer), MediaStore	Playback and media management
Background work	WorkManager, foreground services	Scheduled and long-running tasks
Distribution	Android Studio and Google Play Console	Build, sign and publish (Android App Bundle)

Table 6. *Native development: advantages and disadvantages.*

Advantages	Disadvantages
• Peak runtime performance • Full hardware and OS API access • Best user experience and platform feel • Strong platform security model • Latest OS features available on day one	• Two codebases to maintain • Separate iOS and Android skills required • Higher development cost • Longer time to market

Example. *WhatsApp is built natively on each platform, giving deep integration with the camera, microphone, background notifications and the platform's cryptographic libraries.*

Exercise. *Identify one setting or feature on your own phone that is distinctly iOS or Android, and consider how platform conventions shape what users expect from a native app.*

iOS platform	Android platform
Swift / Objective-C UI (SwiftUI, UIKit)	Kotlin / Java UI (Jetpack Compose, Views)
iOS SDK	Android SDK
iOS kernel	Android kernel
iPhone hardware	Android hardware
Distribution: Apple App Store	Distribution: Google Play

Figure 1. *Native app architecture: one layered stack per platform and two separate codebases.*

4. Cross-Platform App Development

4.1 Definition and core concepts

Cross-platform development lets one codebase produce applications for several platforms. Frameworks follow three strategies: drawing the interface with their own rendering engine, driving the platform's native UI components from a shared language, or sharing business logic while keeping native UI.

Flutter (Google) uses Dart, compiles to native code and draws every pixel with its own engine, Impeller, which is now the default renderer on iOS and Android, giving consistent rendering on every platform. The stable release as of September 2026 is Flutter 3.47 [1]. **React Native** (Meta) describes the interface in JavaScript or TypeScript and maps it to real native views. The New Architecture (Fabric and TurboModules, built on the JavaScript Interface, JSI) became the default in version 0.76 and mandatory from 0.82; Hermes V1 became the default JavaScript engine in 0.84; the current stable release is 0.87 [1, 2]. Expo has become the standard way to start React Native projects. **Kotlin Multiplatform (KMP)** has been stable since November 2023 and shares business logic across Android, iOS, desktop and server while leaving the UI native; with **Compose Multiplatform**, JetBrains also offers shared UI that is stable on Android, iOS and desktop, with web (Wasm) still in beta [3]. **.NET MAUI** (C# and .NET 10) succeeds Xamarin, which reached end of support in May 2024.

Table 7. *Principal cross-platform frameworks (October 2026).*

Framework	Language	UI approach	Notes
Flutter	Dart	Own renderer (Impeller)	Widest platform coverage from one codebase; 3.47 stable
React Native	JavaScript / TypeScript	Native components via Fabric	Largest ecosystem; New Architecture mandatory; 0.87 stable
Kotlin Multiplatform	Kotlin	Native UI, or shared Compose Multiplatform	Shared logic; strong fit for existing Android teams
.NET MAUI	C#	Native controls	Microsoft and Azure ecosystem; Xamarin successor
Lynx (ByteDance)	JavaScript	Own renderer	Newer entrant; smaller ecosystem

Surveys have placed Flutter at roughly 46% of cross-platform developers and React Native at 35–38%, with Kotlin Multiplatform the fastest-growing of the three; these figures vary by survey and year and should be read as indicative [2, 5]. Typical code sharing is 70–90% for Flutter and React Native, and close to 100% of business logic for KMP, with the remainder being platform-specific customisation.

Table 8. *Cross-platform development: advantages and disadvantages.*

Advantages	Disadvantages
• One codebase lowers cost • Faster time to market • Shared business logic • Large developer communities • Near-native performance for most apps	• Platform-specific UX nuances need extra work • Some device APIs need platform plugins • Larger app size • Support for brand-new OS features can lag

Example. *Discord uses React Native for large parts of its mobile apps, shipping features quickly on iOS and Android from shared JavaScript.*

Shared layer
Single codebase: shared business logic and UI components
Framework layer: Flutter (Impeller renderer), React Native (Fabric / JSI), KMP, .NET MAUI
iOS native APIs and kernel | Android native APIs and kernel

Figure 2. *Cross-platform architecture: a shared layer above platform-specific APIs; typically 70–90% code reuse.*

5. Hybrid App Development

5.1 Definition and core concepts

Hybrid applications wrap HTML, CSS and JavaScript in a native container that hosts a WebView. The shell is native, allowing app-store distribution and device access through plugins, but the interface is essentially a web page rendered by an embedded browser component. Apache Cordova, which grew out of PhoneGap, pioneered the approach; Ionic added a component library imitating native patterns; and **Capacitor**, from the Ionic team, is now the modern default, with a cleaner plugin system and good integration with React, Vue and Angular. Capacitor is also a common route for packaging an existing PWA as a store app. Cordova is largely in maintenance and is not recommended for new projects.

Table 9. *Hybrid development: advantages and disadvantages.*

Advantages	Disadvantages
• One codebase for several platforms • Low barrier for web developers • Access to the large web ecosystem • Fast development for simple apps	• Performance ceiling below native and cross-platform • UI may not feel fully native • WebView memory and startup overhead • Plugin dependencies widen the security surface

Example. *Many enterprise internal tools and B2B applications use Ionic and Capacitor, valuing development speed and web-team reuse over peak performance.*

Layer
HTML / CSS / JavaScript app (Angular, React, Vue)
WebView / browser engine
Native container (app shell)
Capacitor / Cordova plugin bridge
iOS APIs | Android APIs

Figure 3. *Hybrid app architecture: a web app inside a native container; device APIs are reached only through the plugin bridge.*

6. Progressive Web Apps (PWAs)

6.1 Definition and core concepts

A Progressive Web App is a web application that uses modern browser APIs to deliver an experience comparable to a native app. It runs in any modern browser, can be installed to the home screen and can work offline through the Service Worker API. The Service Worker is a background JavaScript worker that acts as a programmable network proxy, serving cached responses when the device is offline. Combined with the Web App Manifest, a JSON file defining the app's name, icons, display mode and theme, it provides much of the behaviour that once required native code.

6.2 Platform support

Core APIs are the Service Worker, Web App Manifest, Push, Cache and Notifications APIs, with IndexedDB for structured data. Installability requires HTTPS and a valid manifest; exact criteria vary by browser. Support has improved on iOS: web push works for sites added to the Home Screen from iOS 16.4, and recent iOS releases treat Home Screen sites as web apps by default. Gaps remain, however: Background Sync is largely Chromium-only, Web Bluetooth and Web NFC exist only in some Chromium browsers, and capabilities should be checked per target browser before committing to a PWA.

Table 10. *PWAs: advantages and disadvantages.*

Advantages	Disadvantages
• No app store required • Works on any device with a modern browser • Offline functionality • Installable to the home screen • Automatic updates	• Limited native API access • Browser support varies, especially on iOS • Hardware APIs only in some browsers • Weaker store discoverability

Example. *Twitter Lite, Twitter's PWA, reported up to 70% less data usage and 65% more pages per session in markets with limited connectivity, showing the value of PWAs for global reach.*

Layer
PWA shell (HTML / CSS / JS): user interface
Service Worker (offline, caching, push) | Web App Manifest (install, icons, display mode)
Cache API | Push and Notifications API | IndexedDB
Remote server / CDN over HTTPS: data fetch and background sync

Figure 4. *Progressive Web App layered architecture: installable, offline-capable and served over HTTPS.*

6.3 How Service Workers work

A Service Worker runs in a thread separate from the page and therefore cannot access the DOM. Its power lies in intercepting every network request the page makes and responding from a cache, which is the foundation of offline experiences (Tables 11 and 12).

Table 11. *Service Worker lifecycle.*

Phase	What happens	Developer action
Register	The page calls navigator.serviceWorker.register('sw.js')	Check support; register on page load
Install	The browser downloads and installs the new worker	Pre-cache critical assets in the install event
Waiting	A new version waits until pages using the old one close (unless skipWaiting() is called)	Decide whether to prompt a refresh
Activate	The new worker takes control	Delete old caches in the activate event
Fetch	The worker intercepts requests from its pages	Apply a cache-first or network-first strategy
Idle	The browser may stop an idle worker to save memory	State is not persistent; use IndexedDB
Update	The browser checks for a changed worker file on navigation	Version cache names so old caches can be cleared

Table 12. *Service Worker caching strategies.*

Strategy	How it works	Best for
Cache First	Check the cache; fall back to the network	App shell, fonts, icons, static assets
Network First	Try the network; fall back to the cache offline	API data, feeds, user-generated content
Stale While Revalidate	Serve from cache immediately and refresh in the background	Non-critical content where slight staleness is acceptable
Network Only	Always fetch from the network	Payments, real-time data, analytics
Cache Only	Serve only from the cache	Pre-cached, version-locked assets

Table 13. *Key Web App Manifest fields.*

Field	Example	Effect
name / short_name	My App / MyApp	Full name on the splash screen; label under the icon (about 12 characters)
start_url	/index.html	URL opened on launch from the home screen
display	standalone	Hides browser chrome so the app feels native
background_color / theme_color	#517891 / #57B9FF	Splash background; toolbar or status bar colour
icons	[{src, sizes, type}]	Icons at several resolutions (typically 48–512 px)
orientation / scope	portrait / /app/	Orientation lock; URLs belonging to the PWA

7. Web Applications

7.1 Browser-based architecture

Web applications run entirely in a browser, need no installation, communicate with servers over HTTP(S) APIs and render dynamic content with JavaScript frameworks. Modern web apps blur the line between desktop software and browser tools. Capabilities continue to expand: WebAssembly enables near-native compute, and WebGPU is now available across the major browsers, supporting graphics and on-device AI workloads.

7.2 SPA versus MPA

Table 14. *Single Page Applications and Multi-Page Applications compared.*

	SPA	MPA
Definition	One HTML page, updated dynamically with JavaScript	A separate HTML page per route
Examples	Gmail, Notion, Figma	Amazon, Wikipedia, news sites
Performance	Fast after the initial load	Each navigation needs a server round trip
SEO	Needs extra configuration	Naturally SEO-friendly
Typical tools	React, Vue, Angular	Server rendering (Django, Rails, Laravel)
Complexity	Higher front-end complexity	Simpler mental model

Table 15. *Web applications: advantages and disadvantages.*

Advantages	Disadvantages
• No installation • Works on any device with a browser • Instant updates for every user • Easy deployment across a team	• Limited native hardware access • Needs connectivity (unless a PWA) • Performance ceiling below native

8. Architecture Decision Framework

8.1 Decision factors matrix

Selecting an architecture is a multi-dimensional decision. Table 16 scores each approach on a 1–5 scale where 5 is always the most favourable result (for example, 5 for development cost means the lowest cost). Treat it as a starting framework and adjust the weights to the project's constraints. Kotlin Multiplatform with native UI sits between the native and cross-platform columns.

Table 16. *Architecture decision matrix (1 = least favourable, 5 = most favourable).*

Criterion	Native	Cross-platform	Hybrid	PWA	Web app
Performance	5	4	3	3	3
Development cost (5 = lowest)	1	3	4	5	5
Time to market (5 = fastest)	2	3	4	5	5
UX quality	5	4	3	3	3
Hardware access	5	4	2	2	1
Offline support	5	5	3	4	2
Maintainability (5 = easiest)	2	4	3	4	5
Scalability	4	4	3	4	5
Team skill required (not scored)	High	Medium	Low–Medium	Low	Low

8.2 Performance considerations

Performance is a composite of startup time, frame rate (typically 60 fps or higher, and 90–120 Hz on modern displays), memory, battery and network efficiency. Native apps have the highest ceiling because they are compiled for the platform with direct OS access. Flutter, and React Native since Hermes V1 and the New Architecture, have closed much of the gap for typical applications. Well-optimised native apps cold-start in about a second or less, whereas PWAs and web apps are usually slowest because JavaScript must be downloaded and parsed. Hybrid apps carry WebView memory overhead and are most likely to drop frames on complex screens. Background processing and GPS are most battery-efficient through native APIs, while PWAs rely on Service Workers for offline caching.

9. Real-World Case Studies

The three case studies below show how products match architecture to their most demanding requirements. They are simplified summaries based on public engineering information; details change over time, so treat them as teaching examples rather than exact specifications.

9.1 Instagram: native performance for camera and media

Table 17. *Instagram architecture.*

Component	Approach
Client	Native iOS (Objective-C, Swift) and Android (Java, Kotlin) apps with separate codebases
Camera pipeline	AVFoundation on iOS; Camera2 / CameraX on Android
Media processing	GPU-accelerated filters using Metal on iOS and OpenGL ES or Vulkan on Android
Feed rendering	UICollectionView and RecyclerView for smooth 60 fps scrolling
Backend and CDN	Django (Python) API servers with Cassandra and PostgreSQL among the stores; Meta's global CDN

Lessons. Camera and GPU pipelines are hard to abstract reliably, so native APIs give the most control. Separate codebases allow platform-specific UX such as haptics and gestures. Real-time filters need direct GPU access, which hybrid apps struggle to match. At a user base in the billions, small performance gains justify the investment.

9.2 Uber: real-time GPS and distributed services

Table 18. *Uber architecture.*

Component	Approach
Client	Native iOS (Swift) and Android (Kotlin / Java); React Native was dropped for key screens
Location engine	Fused location data with filtering (for example Kalman filters) for smooth, accurate GPS
Real-time layer	Persistent connections (WebSockets or push channels) keep rider and driver state in sync
Maps	Native map SDKs for custom rendering and routes
Backend	Thousands of microservices (mainly Go and Java) linked by Apache Kafka event streams
Matching	Proprietary geospatial matching supported by hexagonal indexing (H3)

Lessons. Reliable background GPS depends on native location APIs; browsers cannot guarantee it. Framework choices should be revisited as needs grow, as when Uber moved critical screens off React Native. Battery optimisation for background GPS needs platform-specific tuning, and offline resilience benefits from native storage and networking.

9.3 Spotify: native streaming with a shared core

Table 19. *Spotify architecture.*

Component	Approach
Audio engine	Shared C++ core for playback, reused across platforms (NDK / JNI on Android)
Offline playback	Encrypted local storage of downloads, with DRM protection
UI layer	Native toolkits: SwiftUI and UIKit on iOS, Jetpack Compose on Android
Streaming	Adaptive-bitrate audio over HTTP-based protocols
Backend and CDN	Microservices and data pipelines (for example Apache Beam) powering recommendations; global CDN

Lessons. Audio playback is first-class, so a shared C++ engine keeps quality consistent. Offline-first features need native file-system and encryption APIs. The guiding rule is to share what is platform-independent (the core) and keep the UI native, an idea that Kotlin Multiplatform now makes available to a much wider range of teams.

10. Industry Trends and Future Directions

10.1 Server-side rendering and meta-frameworks

Client-side SPAs dominated the mid-2010s, but the industry has moved toward mixed rendering. Meta-frameworks such as Next.js (React, including React Server Components), Nuxt (Vue) and SvelteKit let developers choose per page whether to render on the server, at build time or in the browser, improving performance and SEO (Table 20).

Table 20. *Rendering strategies in modern web and app development.*

Strategy	How it works	Best for	Examples
SSR	HTML generated on the server per request	Dynamic, personalised content	Product pages, dashboards
SSG	HTML pre-built at deploy time	Blogs, documentation, marketing	Static exports, Astro
ISR	Static pages regenerated on a schedule or on demand	Semi-dynamic content, large catalogues	E-commerce, news
CSR	HTML built in the browser with JavaScript	Highly interactive apps	Admin tools, editors
Edge rendering	Rendering at CDN edge locations nearest the user	Low-latency global apps	Vercel Edge, Cloudflare Workers

10.2 Emerging patterns (2024–2026)

Island architecture (Astro, Fresh): pages are static HTML by default, with interactive JavaScript "islands" only where needed, greatly reducing shipped JavaScript. **Edge computing** (Cloudflare Workers, Vercel Edge Functions, Deno Deploy) brings logic closer to users. **Kotlin and Compose Multiplatform** make sharing logic, and increasingly UI, across Android, iOS, desktop and web a mainstream option, with web still in beta [3]. **React Native brownfield integration** (for example through Expo) allows screen-by-screen adoption inside existing native apps [2]. **On-device AI** through platform model APIs lets apps run language and vision models locally, which favours native or well-bridged cross-platform stacks. **AI-assisted development**, through tools such as GitHub Copilot, Cursor and Claude Code, is now embedded in everyday workflows for generation, review and refactoring; frameworks are beginning to ship their own agent tooling, for example MCP servers for Flutter and agent skills for React Native [2]. **Mini-app ecosystems** (WeChat, Alipay, TikTok) form a platform-within-a-platform category, running web-based apps inside a super-app container with access to its APIs.

Note for students. The landscape evolves quickly. The strongest engineers are not those who know every framework, but those who understand the trade-offs and can reason about new technologies from first principles.

Appendix A. Glossary of Key Terms

API. A defined contract that lets software components communicate.

APK / IPA / AAB. Package formats for Android (APK; Google Play uses the Android App Bundle) and iOS (IPA).

Background Sync. A Service Worker API that defers requests until connectivity returns (mainly Chromium).

Bundle size. Total size of an app's code and assets; larger bundles slow downloads and cold starts.

Capacitor. An open-source native runtime from the Ionic team giving web apps access to device APIs; successor to Cordova.

CDN. A geographically distributed network that serves static assets from locations near the user.

Compose Multiplatform. JetBrains' shared declarative UI framework for Kotlin, stable on Android, iOS and desktop.

Dart. Google's language for Flutter; compiles to native code, JavaScript or WebAssembly.

Flutter. Google's UI framework for apps on mobile, web and desktop from a single Dart codebase, rendered by Impeller.

Hermes. The JavaScript engine used by React Native; Hermes V1 became the default in 0.84.

Hybrid app. A web-technology app wrapped in a native shell using a WebView.

Impeller. Flutter's rendering engine, which draws the interface itself.

IndexedDB. A browser API for storing substantial structured data; used by PWAs for offline persistence.

Ionic. A UI toolkit for building mobile and desktop apps with web technologies.

JSI. JavaScript Interface; React Native's native module layer enabling synchronous calls to native code.

Kotlin / KMP. JetBrains' language, Google's preferred choice for Android; Kotlin Multiplatform shares code across platforms.

Microservices. An architecture of small, independently deployable services.

MVC / MVP / MVVM / MVI. Internal patterns separating interface, logic and data (see Chapter 2).

Native app. An app built for one OS with its official SDK, languages and guidelines.

PWA. A web app using Service Workers and a manifest for offline use and installation.

React Native. Meta's framework for native mobile apps with React and JavaScript.

SDK. A collection of tools, libraries and documentation for building on a platform.

Service Worker. A background browser script acting as a network proxy for offline, push and sync features.

SPA. A web app that loads one HTML page and updates content dynamically.

SSR / SSG / ISR / CSR. Strategies deciding where and when HTML is produced (see Table 20).

Swift. Apple's language for iOS, macOS, watchOS, tvOS and visionOS.

TypeScript. A typed superset of JavaScript widely used with React Native, Angular and Ionic.

Web App Manifest. A JSON file defining a PWA's name, icons, start URL, display mode and colours.

WebView. A browser component embedded in a native app to render web content.

Xamarin / .NET MAUI. Microsoft's cross-platform frameworks; MAUI succeeds Xamarin, which ended support in May 2024.

References and Sources

[1] Droids On Roids (2026). *Flutter vs React Native in 2026* (current stable releases as of September 2026); Tech Insider (2026), *React Native vs Flutter 2026*.

[2] Cheesecake Labs (2026). *State of Mobile 2026* (React Native 0.84, Expo SDK 55, Hermes V1); CatDoes (2026), *Flutter vs React Native 2026 Compared*; Cybertize Web (2026), *Mobile App Development Trends Report 2026–2027*.

[3] Dual Media (2026). *Kotlin Multiplatform 2026 vs Flutter vs React Native*, citing JetBrains and Kotlin supported-platforms documentation (status as of May 2026).

[4] Releasebot (September 2026). *iOS and iPadOS 27 release notes*; Apple WWDC 2026 coverage.

[5] Uno Platform (2026). *5 Best Cross Platform Frameworks for App Dev in 2026*; Java Code Geeks (2026), *Kotlin Multiplatform vs. Flutter vs. React Native*.

Secondary sources differ on adoption figures and some version numbers; verify against official framework documentation before relying on them. Base content: *Modern App Development Approaches Handbook*, curated by Angela Guilherme (September 2026 edition).