Chương 1

Chap1

1 Tổng quan về kiến trúc hệ thống

1.1 Tổng quan về kiến trúc hệ thống trong ứng dụng di động

Khái niệm kiến trúc hệ thống: Kiến trúc hệ thống trong phát triển ứng dụng di động bao gồm nhiều thành phần như phần cứng thiết bị, ứng dụng di động và hệ thống server hỗ trợ. Mục tiêu của kiến trúc là tạo ra một hệ thống đồng nhất, trong suốt và khả chuyển.

Cần quan tâm đến kiến trúc hệ thống vì kiến trúc hệ thống:

- Ảnh hưởng trực tiếp đến hiệu suất và khả năng mở rộng của ứng dụng.
- Giúp ứng dụng hoạt động ổn định, hạn chế lỗi phát sinh.
- Dễ dàng phát hiện và xử lý lỗi giữa các thành phần.

1.2 Các yếu tố quan trọng trong kiến trúc hệ thống

Tính đồng nhất:

- Dữ liệu cần có sự nhất quán giữa client và server.
- Sử dụng RESTful API hoặc GraphQL để chuẩn hóa giao tiếp giữa các thành phần.
- Các hệ thống cần đảm bảo dữ liệu được lưu trữ và đồng bộ hóa đúng cách.

Tính trong suốt:

- Các thành phần trong hệ thống cần có khả năng phát hiện lỗi và thông báo rõ ràng.
- Sử dụng các công cụ logging, monitoring như Firebase Crashlytics, Sentry để theo dõi lỗi.
- Cấu trúc hệ thống phải cho phép dễ dàng kiểm tra và sửa lỗi nhanh chóng.

Tính khả chuyển:

- Hệ thống cần có kiến trúc linh hoạt để dễ dàng thay đổi thành phần mà không làm gián đoạn hoạt động.
- Sử dụng microservices hoặc modular architecture để tách biệt các thành phần.
- Ứng dụng công nghệ containerization như Docker, Kubernetes để tăng tính khả chuyển.

1.3 Các mô hình kiến trúc phổ biến trong ứng dụng di động

Hiện nay có một số kiến trúc phổ biến trong ứng dụng di động bao gồm:

- Kiến trúc Layers (3 thành phần): Kiến trúc này chia ứng dụng thành ba lớp chính: lớp giao diện người dùng (UI), lớp logic nghiệp vụ (Business Logic), và lớp dữ liệu (Data). Mô hình này giúp tách biệt các thành phần của ứng dụng, làm cho ứng dụng dễ bảo trì và mở rộng.
- Kiến trúc MVC (Model-View-Controller): Ở kiến trúc này, ứng dụng được phân chia thành ba thành phần chính: Model (dữ liệu và logic), View (giao diện người dùng), và Controller (xử lý sự kiện và giao tiếp giữa Model và View).
- Kiến trúc MVVM (Model-View-ViewModel): Tương tự như MVC nhưng có sự bổ sung của ViewModel, giúp quản lý dữ liệu giữa View và Model, tạo sự tách biệt rõ ràng hơn giữa các thành phần trong ứng dụng.
- Kiến trúc MVI (Model-View-Intent): Kiến trúc này phân chia mỗi hành động người dùng (Intent) thành các sự kiện, gửi đến View, rồi View trả lại trạng thái mới cho Model, giúp duy trì tính nhất quán và đồng bộ trong ứng dụng.

– Kiến trúc Client-Server: Mô hình này chia ứng dụng thành hai phần: Client (ứng dụng di động) và Server (máy chủ). Client gửi yêu cầu đến Server, Server xử lý và trả về kết quả cho Client. Mô hình này thường được sử dụng trong các ứng dụng cần giao tiếp với cơ sở dữ liệu hoặc dịch vụ trực tuyến.

1.4 Công nghệ và công cụ hỗ trợ

Backend cho ứng dụng di động:

- Node.js với NestJS: Node.js là môi trường runtime cho JavaScript, cho phép xử lý các tác vụ bất đồng bộ hiệu quả. NestJS là framework dựa trên Node.js, giúp phát triển các ứng dụng server-side bằng cách sử dụng TypeScript, mang lại sự tổ chức và cấu trúc tốt cho dự án.
- Django: Một framework Python mạnh mẽ, Django cung cấp các công cụ để xây dựng ứng dụng web an toàn và nhanh chóng. Django thường được sử dụng trong các ứng dụng yêu cầu bảo mật cao và khả năng mở rộng tốt.
- Spring Boot: Là framework Java giúp phát triển các ứng dụng backend nhanh chóng và dễ dàng, đặc biệt là các ứng dụng có tính mở rộng cao và yêu cầu quản lý phức tạp.

Database:

- MySQL: Hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở phổ biến, MySQL phù hợp với các ứng dụng yêu cầu tính ổn định cao và khả năng mở rộng.
- PostgreSQL: Hệ quản trị cơ sở dữ liệu quan hệ mạnh mẽ, hỗ trợ các tính năng phức tạp hơn MySQL, phù hợp cho các ứng dụng yêu cầu tính nhất quán và linh hoạt cao.
- Firebase Firestore: Dịch vụ cơ sở dữ liệu NoSQL của Google, cung cấp tính năng đồng bộ thời gian thực, phù hợp cho các ứng dụng di động yêu cầu đồng bộ dữ liệu nhanh chóng và liên tục.

Frontend và Mobile Frameworks:

- React Native: Framework giúp phát triển ứng dụng di động cross-platform sử dụng JavaScript và React, tiết kiệm thời gian phát triển với mã nguồn chung cho cả iOS và Android.

- Flutter: Framework của Google sử dụng ngôn ngữ Dart, giúp xây dựng giao diện người dùng đẹp mắt và mượt mà cho các ứng dụng di động.
- Swift: Ngôn ngữ lập trình dành riêng cho iOS, tối ưu hóa cho các ứng dụng di động trên hệ sinh thái Apple.
- Kotlin: Ngôn ngữ chính để phát triển ứng dụng Android, mang lại cú pháp đơn giản và dễ duy trì, tương thích tốt với Java.

CI/CD và DevOps trong phát triển hệ thống:

- GitHub Actions: Công cụ tự động hóa tích hợp với GitHub, giúp quản lý quy trình CI/CD cho các dự án phần mềm.
- Jenkins: Công cụ tự động hóa mã nguồn mở giúp quản lý các quy trình tích hợp và triển khai liên tục.
- Docker: Công cụ giúp đóng gói và triển khai ứng dụng trong các container, đảm bảo tính nhất quán giữa các môi trường khác nhau.

2 Lịch sử ngành lập trình di động

2.1 Tổng quan về lịch sử ngành lập trình di động

Ngành lập trình di động gắn liền với sự phát triển không ngừng của thiết bị di động – từ những chiếc điện thoại đơn giản chỉ dùng để nghe gọi cho đến các thiết bị thông minh mạnh mẽ như ngày nay. Trong suốt quá trình phát triển, thiết kế của thiết bị di động đã trải qua nhiều giai đoạn, mỗi giai đoạn đều đặt ra những yêu cầu và xu hướng mới cho ngành lập trình.

2.2 Một số thiết kế của điện thoại di động qua từng thời kỳ

Giai đoạn điện thoại dạng "bag" (túi xách):

- Vào những năm 1980, những chiếc điện thoại di động đầu tiên ra đời với thiết kế to, nặng và cần mang theo như một chiếc túi xách. Chúng chủ yếu dùng để nghe gọi, không có giao diện người dùng phức tạp. Lúc này, lập trình di động chưa thực sự tồn tại vì các thiết bị không hỗ trợ cài đặt thêm phần mềm.

Thế hệ điện thoại nhỏ gọn cầm tay:

– Khi công nghệ vi mạch tiến bộ, điện thoại trở nên nhỏ gọn hơn và có thể cầm vừa lòng bàn tay. Các thiết bị này bắt đầu có màn hình đơn sắc và bàn phím vật lý đơn giản. Một số dòng điện thoại bắt đầu hỗ trợ các ứng dụng đơn giản như đồng hồ, lịch hay trò chơi (ví dụ: Snake trên Nokia). Đây là thời điểm lập trình di động bắt đầu hình thành, với các ứng dụng viết bằng ngôn ngữ như C hoặc Java ME.

Thời kỳ điện thoại vỏ gập (flip phone):

– Vào đầu những năm 2000, điện thoại vỏ gập trở nên phổ biến nhờ thiết kế thời trang và tiện lợi. Một số hãng còn tích hợp thêm camera, nhạc chuông đa âm, và khả năng kết nối internet cơ bản (WAP). Lập trình viên lúc này bắt đầu xây dựng các ứng dụng Java nhỏ để chạy trên nền tảng Java ME hoặc Brew.

Sự xuất hiện của điện thoại có bàn phím QWERTY

– BlackBerry là đại diện tiêu biểu cho dòng điện thoại với bàn phím đầy đủ, hướng đến người dùng doanh nhân. Giao diện người dùng lúc này đã phát triển hơn, hỗ trợ email, trình duyệt web và một số ứng dụng văn phòng. Lập trình di động trở nên chuyên nghiệp hơn với SDK riêng cho từng hệ điều hành.

Kỷ nguyên của điện thoại thông minh với màn hình cảm ứng

– Năm 2007, iPhone ra đời đã mở ra một kỷ nguyên mới: điện thoại thông minh với màn hình cảm ứng và rất ít nút vật lý. Android cũng nhanh chóng phát triển sau đó. Đây là bước ngoặt lớn của ngành lập trình di động, khi các nền tảng như iOS và Android cung cấp công cụ phát triển mạnh mẽ, chợ ứng dụng (App Store, Google Play) và cộng đồng lập trình viên đông đảo.

Có thể dễ dàng nhận thấy xu hướng thiết bị hiện đại là: màn hình lớn, đa chức năng. Ngày nay, các thiết bị di động có màn hình lớn, hiệu năng cao, pin lâu và khả năng xử lý mạnh mẽ như một chiếc máy tính thu nhỏ. Một số thiết bị như iPad còn được dùng thay thế laptop và hỗ trợ cả nghe gọi thông qua các ứng dụng. Điều này đặt ra yêu cầu cao cho lập trình viên: ứng dụng phải tối ưu giao diện trên nhiều kích thước màn hình, hỗ trợ đa nhiệm, tiết kiệm pin và bảo mật tốt.

2.3 Motorola DynaTAC 8000X – Bước ngoặt đầu tiên của điện thoại di động

Chiếc Motorola DynaTAC 8000X được ra mắt vào năm 1983, đánh dấu cột mốc quan trọng trong lịch sử viễn thông khi trở thành chiếc điện thoại di động đầu tiên được thương mại hóa. Tuy thô sơ so với tiêu chuẩn ngày nay, DynaTAC đã đặt nền móng cho ngành công nghiệp điện thoại di động và sau này là lập trình di động.

Thông số và đặc điểm nổi bật:

- Năm sản xuất: 1983. Đây là thời điểm đánh dấu sự bắt đầu chính thức của kỷ nguyên điện thoại di động cá nhân.
- Kích thước: 13 x 1.75 x 3.5 inch (33 x 4.4 x 8.9 cm). Với kích thước khá lớn, người dùng thường phải cầm bằng cả hai tay. Thiết kế của máy trông giống như một "cục gạch"nên còn được gọi là “brick phone”.
- Khối lượng: 2.5 pounds (1.1 kg). Rất nặng so với điện thoại hiện đại, chỉ phù hợp với người dùng có nhu cầu đặc biệt hoặc có tài chính dư dả.
- Giá bán: $3,995 vào thời điểm ra mắt. Mức giá này tương đương khoảng hơn 10.000 USD theo tỷ giá điều chỉnh lạm phát hiện nay – biến nó thành một món hàng xa xỉ.
- Chi phí sử dụng: bao gồm 2 loại, phí thuê bao hàng tháng (Monthly fee) và phí theo từng phút gọi (Pay per minute). Người dùng không chỉ trả tiền để sở hữu máy, mà còn phải trả phí sử dụng rất cao để duy trì dịch vụ.

Ý nghĩa của chiếc điện thoại này đối với ngành lập trình di động:

- Tuy Motorola DynaTAC 8000X chưa hỗ trợ các ứng dụng như điện thoại thông minh ngày nay, nhưng sự ra đời của nó đã khởi động một thị trường mới – nơi mà các nhà sản xuất, kỹ sư phần cứng và sau này là lập trình viên di động bắt đầu tham gia để khai phá tiềm năng công nghệ không dây.
- Dù lúc này chưa có khái niệm về lập trình ứng dụng di động, sự ra đời của DynaTAC đã mở ra nhu cầu phát triển các nền tảng và hệ điều hành di động trong tương lai.

2.4 Từ "cục gạch" đến trải nghiệm người dùng hiện đại

Sau khi điện thoại di động ra đời, công nghệ không ngừng phát triển để biến chúng từ những thiết bị to lớn, cồng kềnh và đắt đỏ trở thành vật dụng phổ biến và gần như không thể thiếu trong đời sống hàng ngày. Cùng với đó, nhu cầu về trải nghiệm người dùng (User Experience - UX) trên thiết bị di động ngày càng trở nên quan trọng, kéo theo sự phát triển mạnh mẽ của ngành lập trình di động.

Vì sao trải nghiệm người dùng trở nên quan trọng?

- Điện thoại không còn là mặt hàng xa xỉ. Giá thành của thiết bị di động đã giảm đáng kể, nhờ vào sự cạnh tranh giữa các hãng sản xuất và tiến bộ trong công nghệ. Điều này giúp điện thoại thông minh tiếp cận được đông đảo người dùng ở mọi tầng lớp xã hội. Khi người dùng ngày càng đông, họ bắt đầu có nhiều lựa chọn và kỳ vọng cao hơn đối với chất lượng ứng dụng.

- Công nghệ phần cứng phát triển vượt bậc Thiết bị ngày càng nhỏ gọn, nhẹ, pin lâu hơn, màn hình sắc nét, cảm ứng nhạy, thậm chí có khả năng chống va đập và chống xước tốt. Những cải tiến này tạo điều kiện cho các ứng dụng di động trở nên phong phú, đa dạng về giao diện và chức năng. Khi phần cứng đủ mạnh, việc đầu tư cho trải nghiệm phần mềm trở nên cấp thiết.

Sự thay đổi trong vai trò của các nhà sản xuất phần cứng

- Ban đầu, chính các nhà sản xuất phần cứng cũng là người viết phần mềm cho thiết bị của họ. Tuy nhiên họ không muốn và không thể tiếp tục phát triển phần mềm phù hợp với nhu cầu ngày càng đa dạng của người dùng. Đồng thời, các hãng cũng không sẵn sàng chia sẻ các bí mật công nghệ về phần cứng của mình cho cộng đồng để đảm bảo tính bảo mật và lợi thế cạnh tranh.

  ⇒ Điều này dẫn đến một nhu cầu cấp thiết: phải có một cầu nối giữa phần cứng và phần mềm – giúp các lập trình viên bên ngoài có thể viết ứng dụng cho các thiết bị mà không cần biết quá sâu về phần cứng bên trong.

  ⇒ Sự ra đời của các chuẩn mở. Để giải quyết vấn đề này, các chuẩn chung bắt đầu được hình thành, tạo ra giao diện lập trình ứng dụng (API) hoặc môi trường trung gian giúp kết nối giữa phần cứng và phần mềm. Chuẩn Web trên di động là một trong những hướng đi đầu tiên. Các trình duyệt web di động cho phép người dùng truy cập thông tin mà không cần cài đặt

ứng dụng. Lập trình viên bắt đầu phát triển các trang web tối ưu hóa cho điện thoại, đặt nền móng cho khái niệm "ứng dụng web di động".

2.5 Chuẩn WAP – Khởi đầu cho trình duyệt web trên di động

Khi điện thoại di động bắt đầu được sử dụng phổ biến hơn, nhu cầu truy cập thông tin (như tin tức, thời tiết, thể thao...) ngay trên thiết bị di động cũng xuất hiện. Tuy nhiên, vào thời điểm đó, hạ tầng mạng di động còn rất hạn chế (chậm, không ổn định, băng thông thấp), nên không thể dùng được các trang web HTML như trên máy tính.

WAP – Wireless Application Protocol là gì?

- WAP (Giao thức ứng dụng không dây) là một chuẩn giao tiếp được thiết kế riêng cho các thiết bị di động, nhằm giúp chúng có thể trao đổi dữ liệu với máy chủ web thông qua mạng không dây yếu kém và không ổn định.
- WAP được xem là một phiên bản đơn giản hóa của giao thức HTTP, tối ưu hóa cho môi trường mạng di động thời kỳ đầu (2G, GPRS...).

Các đặc điểm nổi bật của WAP:

- Sử dụng WML (Wireless Markup Language) thay vì HTML: WML được thiết kế nhẹ hơn rất nhiều so với HTML, giúp hiển thị nội dung văn bản trên các thiết bị có màn hình nhỏ, không hỗ trợ hình ảnh phức tạp. Nó hoạt động giống như HTML nhưng bị giới hạn về thẻ và cấu trúc, chỉ đủ để hiển thị nội dung cơ bản như tiêu đề, đoạn văn, liên kết.
- Tối ưu cho mạng yếu: WAP được xây dựng nhằm giảm thiểu dung lượng dữ liệu truyền tải và xử lý dễ dàng hơn trên các thiết bị có cấu hình thấp.

Một số trang web nổi bật ứng dụng WAP:

- CNN – Cung cấp tin tức thời sự trên nền WAP.
- ESPN – Cung cấp tin thể thao, tỷ số trực tiếp cho người dùng di động.

Đây là những ví dụ tiêu biểu cho việc các nhà phát triển ứng dụng web bắt đầu quan tâm đến nền tảng di động, dù hạn chế rất nhiều so với web truyền thống.

⇒ Ý nghĩa của WAP đối với lập trình di động:

- Đặt nền móng đầu tiên cho khái niệm "ứng dụng web di động".
- Tạo ra môi trường để lập trình viên có thể bắt đầu viết ứng dụng cho điện thoại, dù thông qua giao diện web đơn giản.
- Mở ra kỷ nguyên nội dung số trên di động – không chỉ dùng điện thoại để nghe gọi, mà còn để tiếp cận thông tin như trên máy tính.

2.6 Sự phát triển của thanh toán di động và bước ngoặt trong lập trình ứng dụng

Trong quá trình phát triển của điện thoại di động, một trong những yếu tố quan trọng thúc đẩy sự phát triển mạnh mẽ của ứng dụng đó chính là khả năng thanh toán trên thiết bị di động.

Thanh toán qua SMS – hình thức phổ biến đầu tiên:

- Ở giai đoạn đầu, SMS (Short Message Service) không chỉ là phương tiện giao tiếp, mà còn được tận dụng để thực hiện các giao dịch thanh toán đơn giản. Người dùng gửi tin nhắn đến một đầu số dịch vụ (ví dụ như 8xxx, 6xxx...), sau đó được nhận lại một nội dung như nhạc chuông, hình nền, truyện cười... với mức phí cao hơn nhiều lần so với SMS thông thường.
- Những tin nhắn như vậy được gọi là tin nhắn giá trị gia tăng (VAS – Value Added Service). Đây là hình thức kinh doanh dịch vụ nội dung số đầu tiên trên nền tảng di động.

Các hình thức thanh toán khác ra đời:

- Thẻ cào điện thoại: Người dùng có thể nạp tiền qua thẻ cào rồi trừ trực tiếp vào tài khoản để mua dịch vụ hoặc vật phẩm trong ứng dụng.
- Web charging: Một hình thức thanh toán thông qua kết nối internet di động. Khi người dùng truy cập vào một trang web di động và chọn mua dịch vụ, hệ thống sẽ tự động trừ tiền vào tài khoản điện thoại.

Những phương thức này tuy đơn giản nhưng đã mở ra cánh cửa đầu tiên cho thương mại điện tử trên thiết bị di động, đồng thời tạo điều kiện cho lập trình viên phát triển các ứng dụng có khả năng kiếm tiền trực tiếp từ người dùng.

Sự chuyển mình mạnh mẽ của ngành lập trình di động

- Nhu cầu thanh toán di động tăng cao, cùng với lượng người dùng điện thoại không ngừng mở rộng, khiến các công ty lớn như Apple, Google và nhiều công ty công nghệ khác bắt đầu đầu tư mạnh mẽ vào thị trường di động. Họ xây dựng nền tảng hệ điều hành riêng, kho ứng dụng, API thân thiện với lập trình viên, tạo ra một hệ sinh thái hoàn chỉnh cho phát triển ứng dụng.

- Tuy nhiên chỉ riêng Microsoft chậm chân hơn và không kịp bắt kịp làn sóng này – điển hình là sự thất bại của Windows Phone.

- Trước đó, việc lập trình cho di động trên nền J2ME (Java 2 Micro Edition) là một cơn ác mộng với các lập trình viên. Mỗi thiết bị lại có kích thước màn hình, bàn phím, khả năng xử lý khác nhau. Giao diện đơn điệu, không có chuẩn hóa và thiếu các công cụ hỗ trợ phát triển chuyên nghiệp.

- Tuy nhiên, khi phần cứng phát triển vượt bậc (RAM lớn hơn, màn hình cảm ứng, CPU nhanh hơn...), lập trình di động không còn khổ sở như trước nữa. Các nền tảng mới như iOS (Swift, Objective-C) và Android (Java/Kotlin) ra đời, cho phép lập trình viên dễ dàng tạo ra các ứng dụng hấp dẫn, hiện đại và mượt mà hơn bao giờ hết.

2.7 Thị trường di động trên toàn thế giới

Trước năm 2010, thị trường điện thoại di động trên toàn cầu – đặc biệt là tại Việt Nam – đang nằm trong tay một "ông lớn"không thể chối cãi, đó chính là Nokia.

Nokia – Ông vua không ngai của thời kỳ tiền smartphone:

- Từ các dòng điện thoại đen trắng đơn giản cho đến những mẫu máy có màu, có bàn phím QWERTY, hỗ trợ chơi nhạc MP3, chụp hình hay cài game Java... Nokia đã định hình trải nghiệm di động cho cả một thế hệ người dùng.

- Tại Việt Nam, thương hiệu Nokia từng là biểu tượng của sự bền bỉ, dễ dùng và phổ biến, với các dòng máy huyền thoại như Nokia 1100, 6300, N70, N95...

Bên cạnh Nokia, thị phần còn lại được chia cho các hãng như Blackberry, Samsung, HTC, Sony Ericsson... Mỗi nhà sản xuất này lại cố gắng phát triển hệ điều hành riêng hoặc sử dụng một nền tảng khác biệt để tạo lợi thế cạnh tranh:

- Symbian (Nokia)
- Blackberry OS (Blackberry)
- Windows Mobile, Windows CE (Microsoft)
- Palm OS, Linux Mobile, Bada OS (Samsung)

Tuy nhiên, điểm chung của các hệ điều hành thời kỳ này là thiếu sự nhất quán và không thân thiện với lập trình viên:

- Giao diện không đồng bộ, trải nghiệm người dùng không ổn định.
- Tài liệu phát triển thiếu thốn, công cụ lập trình rối rắm.
- Không có kho ứng dụng tập trung – việc phân phối ứng dụng rất khó khăn.

  ⇒ Chính vì vậy, dù hệ điều hành liên tục được tạo ra và chết yểu, các nhà phát triển phần mềm lúc đó vẫn còn thờ ơ và đủng đỉnh, vì chưa có động lực hay hệ sinh thái rõ ràng để đầu tư nghiêm túc.

Bước ngoặt chỉ tới và thay đổi hoàn toàn khi iPhone xuất hiện.

- Năm 2007, Apple ra mắt iPhone thế hệ đầu tiên, và đến năm 2008, App Store chính thức được giới thiệu.
- Đây là bước ngoặt làm thay đổi toàn bộ ngành công nghiệp di động.
- iPhone mang đến trải nghiệm cảm ứng mượt mà, giao diện hiện đại, không bàn phím vật lý, tạo ra cuộc cách mạng về thiết kế smartphone.
- App Store là nơi tập trung ứng dụng đầu tiên, vừa giúp người dùng dễ tiếp cận phần mềm, vừa mở ra cơ hội kiếm tiền cho lập trình viên.
- Bộ công cụ phát triển (SDK) dành cho iOS rất mạnh mẽ, tài liệu rõ ràng, cộng đồng lớn, khiến việc lập trình trở nên hấp dẫn hơn bao giờ hết.

  ⇒ Từ đây, ngành lập trình di động bắt đầu bước sang một kỷ nguyên mới – nơi trải nghiệm người dùng, hiệu năng phần mềm và hệ sinh thái là chìa khóa thành công.

2.8 Thị trường và sự cạnh tranh trong ngành lập trình di động

Ba ông lớn từng thống trị ngành di động: Microsoft – Apple – Google. Khi thị trường di động bắt đầu bùng nổ, ba cái tên lớn nhất cùng lúc định hình nên cuộc đua tam mã trong cả phần cứng, hệ điều hành và hệ sinh thái ứng dụng: Microsoft, Apple và Google.

Microsoft – Tham vọng lớn, nhưng chậm chân:

- Sau khi nhận ra sự bùng nổ của smartphone, Microsoft quyết định đặt cược lớn vào mảng di động, bằng cách mua lại bộ phận sản xuất điện thoại của Nokia vào năm 2013.
- Họ kỳ vọng sẽ thống nhất mọi nền tảng phần mềm dưới một triết lý chung gọi là Windows Universal Platform (UWP) – nghĩa là một ứng dụng có thể chạy được trên cả PC, tablet, lẫn điện thoại.
- Tuy nhiên, Windows Phone xuất hiện quá muộn so với iOS và Android. Kho ứng dụng thì nghèo nàn, cộng đồng lập trình viên ít mặn mà. Mặc dù trải nghiệm người dùng có phần mượt mà nhưng thiếu sự linh hoạt và cá nhân hóa.

  ⇒ Kết quả là Windows Phone dần bị khai tử, để lại bài học đắt giá về tốc độ thích nghi trong thời đại công nghệ thay đổi nhanh chóng.

Apple – Người dẫn đầu cả về trải nghiệm và hệ sinh thái

- Apple từ lâu đã nổi tiếng với triết lý thiết kế "từ trong ra ngoài"– tức là tự sản xuất cả phần cứng và phần mềm, từ đó kiểm soát chặt chẽ chất lượng sản phẩm.
- Họ tạo ra một hệ sinh thái khép kín và nhất quán. iPhone với thiết kế cao cấp, mượt mà. Hệ điều hành iOS tối ưu cao, ít lỗi, bảo mật tốt. App Store với quy trình kiểm duyệt nghiêm ngặt, đảm bảo chất lượng ứng dụng.
- Ngoài ra, Apple còn biết cách thu hút lập trình viên bằng các công cụ mạnh mẽ như Xcode, Swift và hệ thống tài liệu phát triển phong phú.

  ⇒ Cho đến nay, Apple vẫn giữ vị thế hàng đầu trong mảng smartphone cao cấp và có tỷ lệ người dùng trung thành cao nhất.

Google – Người chơi hệ mở và "ông trùm"Android

- Không tự sản xuất phần cứng đại trà như Apple, Google chọn cách xây dựng hệ điều hành Android theo triết lý mã nguồn mở.
- Android nhanh chóng được các hãng sản xuất phần cứng trên toàn thế giới (Samsung, Oppo, Xiaomi, Huawei...) áp dụng rộng rãi, khiến nó trở thành nền tảng phổ biến nhất toàn cầu.
- Mặc dù Android ban đầu bị đánh giá là phân mảnh và thiếu ổn định, nhưng theo thời gian Google đã cải thiện hệ điều hành liên tục qua các phiên bản. Cửa hàng ứng dụng Google Play cũng dần trở nên nghiêm ngặt hơn về nội dung. Các thiết bị Pixel do chính Google sản xuất giúp Google kiểm soát tốt hơn trải nghiệm người dùng.

  ⇒ Nhờ vào thế mạnh về công nghệ, dữ liệu và hệ sinh thái khổng lồ (Gmail, Google Maps, YouTube...), Google vẫn duy trì được vị thế vững chắc trong ngành di động toàn cầu.

⇒Ba ông lớn, ba con đường phát triển – nhưng chính sự cạnh tranh đó đã thúc đẩy ngành lập trình di động trở nên đa dạng, sôi động và phát triển mạnh mẽ như hiện nay.

2.9 Hai hệ điều hành phổ biến nhất ngày nay: iOS và Android.

Hệ điều hành iOS:

- iOS được phát triển và quản lý bởi Apple, được sử dụng trên các thiết bị di động của hãng như iPhone, iPad, và iPod Touch. iOS được biết đến với tính ổn định, hiệu suất cao và bảo mật chặt chẽ.
- Trước đây, lập trình iOS chủ yếu sử dụng Objective-C, một ngôn ngữ được Apple phát triển cho hệ điều hành của mình. Tuy nhiên, từ năm 2014, Apple đã giới thiệu Swift, một ngôn ngữ lập trình hiện đại, dễ học và hiệu quả hơn so với Objective-C, dành cho việc phát triển ứng dụng iOS. XCode là môi trường phát triển tích hợp (IDE) chính thức do Apple cung cấp để lập trình ứng dụng trên iOS.

- Apple cung cấp một số framework mạnh mẽ như UIKit và SwiftUI để xây dựng giao diện người dùng và quản lý các tính năng ứng dụng. Bên cạnh đó, các công cụ như Xcode cho phép lập trình viên kiểm tra, mô phỏng, và tối ưu hóa ứng dụng cho các thiết bị của Apple.
- iOS có triết lý "đóng", nghĩa là Apple quản lý chặt chẽ toàn bộ hệ sinh thái của mình. Điều này bao gồm cả việc phân phối ứng dụng thông qua App Store, nơi các ứng dụng phải trải qua quá trình kiểm duyệt nghiêm ngặt trước khi được phép phát hành. Điều này giúp đảm bảo chất lượng và bảo mật cho người dùng.
- Mặc dù iOS chủ yếu sử dụng các công cụ của Apple, nhưng cũng có các framework đa nền tảng như Flutter, React Native, Xamarin và Unity giúp lập trình viên xây dựng ứng dụng chạy trên cả iOS và Android mà không cần viết mã riêng cho từng hệ điều hành.

Hệ điều hành Android:

- Android là một hệ điều hành mã nguồn mở, được phát triển bởi Google và dựa trên nền tảng Linux. Android được sử dụng rộng rãi trên các thiết bị của nhiều nhà sản xuất khác nhau, chẳng hạn như Samsung, HTC, Sony, và nhiều hãng khác. Điều này tạo ra sự đa dạng về phần cứng, cho phép người dùng có nhiều lựa chọn hơn khi chọn thiết bị di động.
- Lập trình Android chủ yếu sử dụng Java và Kotlin. Java đã là ngôn ngữ chính cho phát triển Android trong nhiều năm, nhưng từ 2017, Google chính thức công nhận Kotlin là ngôn ngữ chính cho lập trình Android, nhờ tính năng hiện đại và sự dễ dàng trong việc tích hợp với Java.
- Android Studio là môi trường phát triển chính thức được Google cung cấp, hỗ trợ các công cụ mạnh mẽ để lập trình viên có thể thiết kế giao diện, kiểm tra ứng dụng trên các thiết bị ảo, và tối ưu hóa ứng dụng cho các phiên bản Android khác nhau. Android Studio tích hợp nhiều công cụ như Android Emulator, Gradle, và Firebase để cải thiện quy trình phát triển.
- Một trong những điểm mạnh của Android là tính mở của nó. Hệ điều hành Android là mã nguồn mở, cho phép các nhà sản xuất khác nhau, như Samsung, Huawei, và Xiaomi, tùy chỉnh hệ điều hành này theo nhu cầu của họ. Điều này dẫn đến một sự đa dạng lớn trong các phiên bản Android, từ giao diện người dùng đến các tính năng bổ sung.

– Android hỗ trợ nhiều framework đa nền tảng như Flutter, React Native, Xamarin, và Unity. Những công cụ này cho phép lập trình viên viết ứng dụng một lần và triển khai trên cả iOS và Android, tiết kiệm thời gian và công sức.

Các nền tảng đa nền tảng:

– Bên cạnh iOS và Android, các công cụ lập trình đa nền tảng như Flutter, Xamarin, React Native, và Unity đang ngày càng phổ biến. Những công cụ này cho phép lập trình viên phát triển ứng dụng một lần và chạy trên nhiều hệ điều hành, giúp tiết kiệm thời gian phát triển và chi phí duy trì. Mặc dù chúng có những ưu điểm về hiệu quả, nhưng vẫn có một số hạn chế về hiệu suất và tính tương thích so với việc phát triển ứng dụng gốc (native) trên từng hệ điều hành.

– Flutter (do Google phát triển) cho phép lập trình viên viết ứng dụng bằng Dart và cung cấp giao diện người dùng mượt mà với hiệu suất cao.

– React Native (do Facebook phát triển) giúp lập trình viên viết ứng dụng bằng JavaScript và tái sử dụng các component từ ứng dụng web.

– Xamarin (do Microsoft phát triển) sử dụng C# và giúp lập trình viên xây dựng ứng dụng gốc cho cả iOS và Android.

– Unity chủ yếu được sử dụng để phát triển trò chơi, nhưng cũng có thể được dùng để tạo ứng dụng di động bằng ngôn ngữ C#.

⇒ Nhìn chung, sự phát triển của các hệ điều hành di động và các công cụ phát triển đa nền tảng đã giúp ngành lập trình di động trở thành một lĩnh vực sôi động và đầy tiềm năng. 4o mini

3 Tổng quan về các kiến trúc phần mềm

Trong lĩnh vực phát triển ứng dụng di động, kiến trúc phần mềm đóng vai trò vô cùng quan trọng. Việc lựa chọn mô hình kiến trúc phù hợp giúp lập trình viên tổ chức mã nguồn một cách hợp lý, dễ bảo trì và mở rộng về sau. Hai hệ điều hành phổ biến nhất hiện nay – iOS và Android – tuy có nhiều điểm khác nhau, nhưng đều khuyến khích sử dụng các mẫu kiến trúc phần mềm để nâng cao hiệu quả phát triển ứng dụng.

3.1 Mẫu kiến trúc Layers

Cả hai hệ điều hành đều cổ vũ lập trình viên áp dụng một số mẫu kiến trúc chung nhằm đảm bảo cấu trúc rõ ràng, dễ kiểm soát. Một trong những mẫu phổ biến là Layers (phân lớp). Theo mô hình này, phần mềm được chia thành ba lớp chính:

- Presentation Layer (Lớp giao diện): Giao tiếp trực tiếp với người dùng, hiển thị thông tin và nhận tương tác.
- Business Logic Layer (Lớp xử lý nghiệp vụ): Chứa logic của ứng dụng, xử lý các thao tác từ người dùng, thực hiện tính toán, kiểm tra dữ liệu.
- Data Layer (Lớp dữ liệu): Quản lý dữ liệu, truy xuất từ cơ sở dữ liệu nội bộ hoặc từ máy chủ.

⇒ Ưu điểm của mô hình này là dễ bảo trì, dễ kiểm thử, giảm sự phụ thuộc giữa các thành phần. Mỗi lớp chỉ thực hiện một nhiệm vụ cụ thể, giúp mã nguồn rõ ràng và dễ quản lý, đặc biệt trong các dự án lớn.

3.2 Mẫu kiến trúc MVC

Trên nền tảng iOS, MVC là mô hình kiến trúc lâu đời và phổ biến. Các lập trình viên iOS thường ưa chuộng mô hình này hơn. Nó phân chia phần mềm thành ba thành phần chính:

- Model: Quản lý dữ liệu, logic xử lý và các quy tắc nghiệp vụ.
- View: Hiển thị thông tin cho người dùng, như giao diện ứng dụng.
- Controller: Là cầu nối giữa Model và View. Nó nhận dữ liệu từ Model và cập nhật cho View, đồng thời xử lý các tương tác từ người dùng.

⇒ Mô hình này đơn giản, dễ hiểu và dễ áp dụng, đặc biệt với những người mới bắt đầu học lập trình iOS. Tuy nhiên, một vấn đề lớn thường gặp trong MVC của iOS là Controller thường bị "phình to", xử lý quá nhiều logic, khiến mã khó đọc, khó bảo trì. Đây được gọi là hiện tượng “Massive View Controller” – một trong những lý do khiến nhiều lập trình viên hiện đại tìm đến các mô hình thay thế như MVVM hoặc VIPER.

3.3 Mô hình kiến trúc MVVM

Khác với nền tảng iOS, nền tảng Android lại ưu tiên sử dụng mô hình MVVM, phù hợp hơn với kiến trúc hiện đại của Android và các công cụ hỗ trợ như LiveData, ViewModel, và Data Binding.

- Model: Cũng giống kiến trúc MVC, thành phần này quản lý dữ liệu và logic xử lý.
- View: Giao diện người dùng, hiển thị dữ liệu từ ViewModel.
- ViewModel: Trung gian giữa View và Model. Nó không chứa tham chiếu trực tiếp đến View, nhưng nhờ cơ chế observer pattern (theo dõi), dữ liệu thay đổi trong ViewModel sẽ được cập nhật tự động lên View.

⇒ MVVM giúp giảm mạnh sự phụ thuộc giữa UI và logic nghiệp vụ. Nhờ đó, View trở nên "mỏng", chỉ đảm nhiệm việc hiển thị, trong khi mọi logic đều được xử lý trong ViewModel. Điều này giúp dễ kiểm thử, tái sử dụng code và phát triển theo nhóm.

3.4 Hỗ trợ mô hình Client – Server

Bên cạnh kiến trúc bên trong ứng dụng, cả iOS và Android đều hỗ trợ xây dựng ứng dụng dựa trên mô hình Client – Server.

- Ứng dụng hoạt động như Client, gửi các yêu cầu HTTP (GET, POST, PUT, DELETE...) đến một Server.
- Server xử lý yêu cầu, truy xuất dữ liệu từ cơ sở dữ liệu, sau đó phản hồi về Client dưới dạng dữ liệu JSON hoặc XML.

Cả hai nền tảng đều cung cấp thư viện hỗ trợ như:

- iOS: URLSession, Alamofire
- Android: Retrofit, OkHttp

⇒ Mô hình Client – Server tạo điều kiện cho ứng dụng hoạt động linh hoạt, nhẹ, dễ cập nhật và đồng bộ dữ liệu giữa nhiều thiết bị người dùng khác nhau.

3.5 Đánh giá tổng quan

Nhìn chung, dù cùng hướng tới việc xây dựng phần mềm có cấu trúc rõ ràng và dễ bảo trì, iOS và Android lại có những lựa chọn mô hình kiến trúc khác nhau. iOS truyền thống với MVC, tuy đơn giản nhưng dễ bị quá tải ở phần Controller, trong khi Android hiện đại hóa với MVVM, tách biệt rõ ràng các thành phần, tận dụng được sức mạnh của các thư viện hỗ trợ. Cả hai nền tảng cũng đều hỗ trợ trao đổi dữ liệu theo mô hình Client – Server, phù hợp với nhu cầu xây dựng ứng dụng kết nối mạng ngày nay. Việc hiểu rõ ưu – nhược điểm của từng mô hình sẽ giúp lập trình viên lựa chọn giải pháp kiến trúc phù hợp, nâng cao hiệu quả phát triển ứng dụng.

4 Phân tích chi tiết các kiến trúc phần mềm

4.1 Kiến trúc phần mềm ba tầng (Three-tier architecture)

Đây là một mô hình tổ chức phần mềm phổ biến, đặc biệt phù hợp với các hệ thống lớn, có khả năng mở rộng và bảo trì lâu dài. Mô hình này phân chia rõ ràng trách nhiệm của từng tầng, từ việc hiển thị, xử lý logic cho đến lưu trữ dữ liệu. Việc áp dụng kiến trúc ba tầng giúp ứng dụng dễ bảo trì, linh hoạt trong mở rộng và tăng khả năng tái sử dụng mã nguồn.

Kiến trúc ba tầng có thể áp dụng cho nhiều loại ứng dụng khác nhau: từ ứng dụng đơn giản đến phức tạp, từ ứng dụng độc lập đến ứng dụng kết nối mạng. Việc tách biệt ba tầng không chỉ làm cho mã nguồn trở nên rõ ràng hơn mà còn cho phép các nhóm phát triển làm việc độc lập trên từng tầng.

Tầng trình diễn (Presentation Layer), đây là tầng giao tiếp với người dùng. Chức năng chính của tầng này bao gồm:

- Hiển thị dữ liệu từ tầng nghiệp vụ theo giao diện trực quan.
- Nhận lệnh từ người dùng (qua các nút bấm, form, tương tác giao diện).
- Không xử lý logic nghiệp vụ, nhờ vậy giao diện có thể dễ dàng tái sử dụng, thay đổi hoặc cập nhật mà không ảnh hưởng đến các tầng khác.

  ⇒ Một lợi ích lớn là khả năng "lắp ghép" lại với các tầng nghiệp vụ khác nhau – giúp cùng một giao diện có thể sử dụng cho nhiều phiên bản khác nhau của hệ thống.

Tầng nghiệp vụ (Business Logic Layer), tầng này giữ vai trò trung tâm trong hệ thống. Nó thực hiện:

- Chuẩn bị dữ liệu đầu vào để gửi đến tầng dữ liệu.
- Chuyển đổi, xử lý dữ liệu nhận về để trả lại cho tầng trình diễn.
- Xử lý các lỗi logic hoặc lỗi phản hồi từ tầng dữ liệu.
- Áp dụng các quy tắc nghiệp vụ, như kiểm tra hợp lệ, xử lý quy trình.

  ⇒ Tầng này giúp cô lập các xử lý phức tạp khỏi giao diện và dữ liệu, đảm bảo khả năng kiểm thử và bảo trì cao.

Tầng dữ liệu (Data Layer), là nơi lưu trữ các thông tin quan trọng nhất của ứng dụng:

- Lưu trữ cơ sở dữ liệu (SQL, NoSQL...).
- Thực hiện các truy vấn để đảm bảo hiệu năng và độ chính xác cao.
- Có thể tích hợp với cơ sở dữ liệu từ xa (server), hệ thống lưu trữ đám mây hoặc tệp cục bộ.

  ⇒ Việc tối ưu tầng dữ liệu giúp cải thiện đáng kể hiệu năng của toàn hệ thống, đặc biệt là trong các ứng dụng có lượng dữ liệu lớn.

⇒ Kiến trúc ba tầng mang lại lợi ích lớn về mặt tổ chức mã nguồn, bảo trì, kiểm thử và phát triển theo nhóm. Mỗi tầng có trách nhiệm riêng, từ đó giúp ứng dụng dễ mở rộng và thích ứng với thay đổi trong tương lai.

4.2 Kiến trúc MVC (Model – View – Controller)

MVC (Model – View – Controller) là một mẫu kiến trúc phần mềm cổ điển, phổ biến trong phát triển ứng dụng, đặc biệt là trên nền tảng iOS. Mục tiêu chính của kiến trúc MVC là tách biệt rõ ràng giữa dữ liệu, giao diện và điều khiển xử lý, từ đó giúp ứng dụng dễ bảo trì, mở rộng và nâng cao trải nghiệm người dùng.

Mô hình này được hình dung như một sơ đồ ba thành phần, mỗi thành phần đảm nhận một vai trò cụ thể:

- Model: Dữ liệu và logic xử lý dữ liệu.

– View: Giao diện hiển thị cho người dùng.

– Controller: Bộ điều phối, tiếp nhận hành động từ người dùng và điều khiển luồng xử lý.

⇒ Ba thành phần hoạt động tách biệt nhưng liên kết chặt chẽ, đảm bảo ứng dụng vận hành trơn tru và dễ dàng điều chỉnh một phần mà không ảnh hưởng đến phần còn lại.

Model – Mô hình dữ liệu:

– Định danh những gì cần trả về cho người dùng.

– Đây là nơi chứa dữ liệu thô, các quy tắc nghiệp vụ và các thao tác xử lý dữ liệu.

– Ví dụ: trong một ứng dụng bán hàng, Model chứa thông tin sản phẩm, đơn hàng, người dùng...

Controller – Bộ điều khiển:

– Tiếp nhận các yêu cầu từ người dùng (qua View).

– Thực hiện các truy vấn tài nguyên, gọi các phương thức xử lý trong Model.

– Là "bộ não" điều phối mọi hoạt động trong ứng dụng.

View – Giao diện người dùng:

– Hiển thị dữ liệu dưới dạng dễ hiểu, thân thiện với người dùng.

– View không xử lý logic nghiệp vụ, mà chỉ phản hồi lại theo những gì Controller và Model cung cấp.

– View sẽ cập nhật nội dung mỗi khi Model thay đổi.

MVC giúp tách biệt rõ ràng chức năng, dễ dàng phát triển, kiểm thử và bảo trì. Đồng thời cho phép nhiều lập trình viên làm việc song song: người thiết kế giao diện làm View, lập trình viên backend làm Model, còn Controller kết nối hai phần này. Ngoài ra, nó còn có tính tái sử dụng mã nguồn tốt khi một Model có thể được dùng cho nhiều View khác nhau.

⇒ Mẫu kiến trúc MVC là một giải pháp hiệu quả giúp tổ chức ứng dụng một cách khoa học và linh hoạt. Việc phân chia ứng dụng thành ba thành phần rõ ràng giúp giảm độ phức tạp khi mở rộng, dễ bảo trì, đồng thời nâng cao hiệu quả làm việc nhóm trong quá trình phát triển ứng dụng. Với iOS, MVC vẫn là lựa chọn được ưa chuộng và hỗ trợ tốt trong môi trường phát triển Xcode và Swift.

4.3 Kiến trúc MVVM (Model - View - ViewModel)

Kiến trúc MVVM (Model – View – ViewModel) là một trong những mẫu thiết kế hiện đại, được áp dụng phổ biến trong phát triển ứng dụng Android (đặc biệt là với sự hỗ trợ từ Jetpack và Kotlin). MVVM ra đời nhằm tối ưu quá trình phát triển ứng dụng bằng cách tách biệt logic hiển thị và logic xử lý, đồng thời tăng tính tự động hóa trong việc cập nhật dữ liệu nhờ cơ chế Data Binding.

MVVM có cấu trúc gần giống với MVC, nhưng thay vì để Controller điều khiển toàn bộ luồng xử lý, MVVM đưa vào một tầng trung gian là ViewModel – chịu trách nhiệm "kết nối thông minh" giữa dữ liệu (Model) và giao diện (View):

- Tương tự như MVC, View hiển thị dữ liệu, Model chứa dữ liệu và logic xử lý.
- Tuy nhiên, Controller được thay thế bằng ViewModel, giúp giảm bớt sự ràng buộc giữa View và Model.

Model – Dữ liệu và logic xử lý:

- Là nơi lưu trữ các dữ liệu chính của ứng dụng (như thông tin người dùng, sản phẩm...).
- Xử lý các nghiệp vụ như tính toán, truy xuất dữ liệu từ cơ sở dữ liệu hoặc API.
- Model không trực tiếp liên hệ với View, mà thông qua ViewModel.

View – Giao diện hiển thị:

- Là phần người dùng tương tác trực tiếp (giao diện ứng dụng).
- View trong MVVM không xử lý logic nghiệp vụ mà chỉ phản ánh lại các dữ liệu từ ViewModel.
- Nhờ vào Data Binding, View có thể tự động cập nhật khi dữ liệu trong ViewModel thay đổi – giúp giảm mã lặp và tăng hiệu suất phát triển.

ViewModel – Cầu nối thông minh:

- Chứa các Model và chuẩn bị dữ liệu để hiển thị cho View.

– Tạo ra các LiveData hoặc Observable để View có thể theo dõi và tự động cập nhật giao diện khi dữ liệu thay đổi.

– Đồng thời, ViewModel cũng xử lý việc truyền dữ liệu từ View sang Model, giúp cập nhật ngược lại khi người dùng nhập liệu hoặc thực hiện thao tác.

Một điểm mạnh nổi bật của MVVM là cơ chế Data Binding – liên kết dữ liệu hai chiều:

– Khi một đối tượng thuộc nhóm View (như EditText) thay đổi, dữ liệu tương ứng trong ViewModel (hoặc Model) cũng tự động cập nhật.

– Ngược lại, khi ViewModel thay đổi giá trị, View cũng cập nhật lại ngay lập tức.

⇒ Điều này giúp hạn chế lỗi khi cập nhật giao diện và rút ngắn thời gian phát triển, đặc biệt là trong các ứng dụng có nhiều thao tác tương tác dữ liệu.

⇒ MVVM là một mô hình kiến trúc mạnh mẽ, phù hợp với các ứng dụng hiện đại cần cập nhật giao diện linh hoạt, liên tục. Với sự hỗ trợ từ Data Binding và LiveData (trong Android), ViewModel giúp đơn giản hóa việc xử lý dữ liệu và đồng bộ giao diện, đồng thời giảm sự phụ thuộc giữa các thành phần, nâng cao khả năng bảo trì và mở rộng về sau. MVVM hiện là lựa chọn ưu tiên trong các dự án Android có quy mô từ vừa đến lớn.

4.4 Kiến trúc Client/Server

Trong thời đại số, đa số các ứng dụng cần trao đổi dữ liệu qua Internet. Mô hình Client/Server trở thành kiến trúc không thể thiếu, đặc biệt với các ứng dụng có tính năng đồng bộ dữ liệu, chia sẻ thông tin theo thời gian thực, hoặc sử dụng tài nguyên trên máy chủ từ xa. Đồng thời, cần kết hợp với các kiến trúc nội bộ như MVC hoặc MVVM để tối ưu hóa việc xây dựng giao diện và xử lý logic.

Kiến trúc Client/Server mô tả mô hình trong đó ứng dụng Client (thiết bị người dùng) gửi yêu cầu đến Server (máy chủ từ xa), thường qua HTTP Request, WebSocket, hoặc Web Service. Các chức năng chính bao gồm:

– Client: Gửi yêu cầu (request), hiển thị dữ liệu, tương tác với người dùng.

– Server: Xử lý yêu cầu, truy cập cơ sở dữ liệu, trả về dữ liệu kết quả.

⇒ Kiến trúc này phù hợp cho các ứng dụng có nhiều người dùng, cần chia sẻ dữ liệu như mạng xã hội, ứng dụng ngân hàng, thương mại điện tử...

5 Các yếu tố ảnh hưởng đến chi phí phát triển ứng dụng

5.1 Các yếu tố chính:

Features and Functionality: Tính năng càng phức tạp, chi phí càng cao.

App Infrastructure: Liên quan đến cách tổ chức hệ thống backend, API, hệ thống lưu trữ, bảo mật, quản lý hiệu năng,...

App Complexity: Mức độ phức tạp về luồng xử lý, logic kinh doanh.

UX/UI Design: Thiết kế giao diện thân thiện, hiện đại cần đầu tư nhiều hơn.

App Maintenance: Chi phí bảo trì sau khi triển khai.

App Security: Yêu cầu bảo mật cao làm tăng chi phí.

App Category: Tùy theo ứng dụng thuộc lĩnh vực nào (y tế, tài chính, mạng xã hội...) sẽ có độ phức tạp khác nhau.

# of Platforms & Pages: Càng nhiều nền tảng (iOS/Android/Web), giao diện càng đa dạng → chi phí tăng.

Location of the Development Team: Địa điểm nhóm phát triển quyết định đến chi phí nhân công.

⇒ Vai trò then chốt của kiến trúc phần mềm - App Infrastructure (Hạ tầng ứng dụng):

- Là xương sống của toàn bộ hệ thống phần mềm, quyết định khả năng mở rộng, hiệu suất, bảo mật, tích hợp dịch vụ khác (API, cơ sở dữ liệu, cloud...).
- Việc thiết kế hạ tầng tốt từ đầu giúp giảm chi phí bảo trì về sau và tăng độ ổn định của ứng dụng.

5.2 Chi phí thuê nhân sự theo khu vực địa lý:

Bảng trên thể hiện khoản chi phí thuê theo giờ cho các vai trò chuyên môn trong ngành công nghệ thông tin, dựa trên thống kê từ 4 khu vực là United States (Hoa Kỳ), Latin America (Mỹ Latin), Eastern Europe (Đông Âu) và Asia (Châu Á). Đồng thời bao gồm 12 vai trò phổ biến trong các dự án phần mềm.

Nhóm Quản lý và Thiết kế Kiến trúc:

- Nhóm này bao gồm 3 vai trò Business Analyst, Architect và Project Manager.
- Đây là nhóm có mức phí cao nhất, đặc biệt tại Hoa Kỳ (lên đến gần $300/giờ).
- Mức giá giảm dần theo khu vực: US > Latin America > Eastern Europe > Asia
- Châu Á có mức phí thấp nhất, từ $30 – $48/giờ, khiến khu vực này hấp dẫn với các công ty thuê ngoài (outsourcing).

  ⇒ Đây là các vị trí quyết định kiến trúc phần mềm, quy trình quản lý và kết nối giữa yêu cầu kinh doanh với kỹ thuật, nên đòi hỏi kinh nghiệm và kỹ năng cao.

Nhóm Lập trình viên (Developer):

- Nhóm này được chia theo kinh nghiệm của lập trình viên, bao gồm Jr. Developer, Mid-Level Dev, Sr. Developer và Lead Developer
- Mức lương tăng dần theo cấp bậc từ Jr. → Mid → Sr. → Lead.
- Sr. Developer và Lead Developer có chi phí gần tiệm cận nhóm kiến trúc sư tại một số khu vực.
- Asia tiếp tục là nơi có mức chi phí thấp nhất ở mọi cấp độ.

Nhóm Kiểm thử phần mềm (QA):

- Nhóm này cũng được chia dựa trên kinh nghiệm, bao gồm Junior QA, Mid-Level QA và Senior QA.

- QA ít tốn kém hơn Developer hoặc Architect, nhưng vẫn có mức chênh lệch đáng kể giữa các khu vực.
- Sr. QA tại Mỹ có thể lên đến $169/giờ, cao hơn cả Mid-Level Dev tại Mỹ ($140).

Thiết kế giao diện (Graphic Designer):

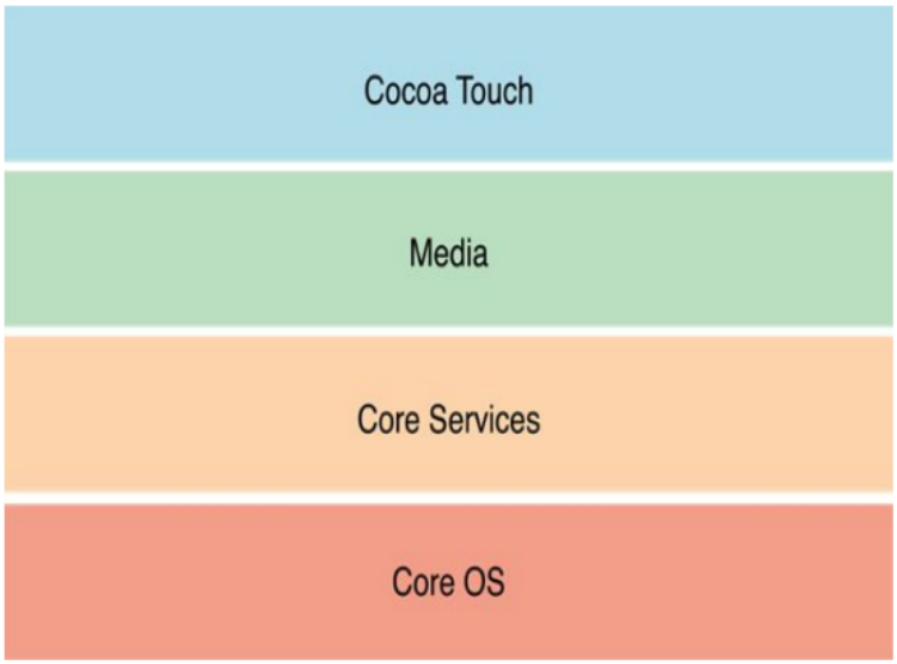
- Vai trò thiết kế tuy không chuyên sâu kỹ thuật nhưng vẫn chiếm chi phí tương đối, đặc biệt ở các thị trường phát triển như Mỹ.

Chương 2

IOS

1 Nền tảng iOS và các thành phần cơ bản

1.1 Kiến trúc tầng của iOS

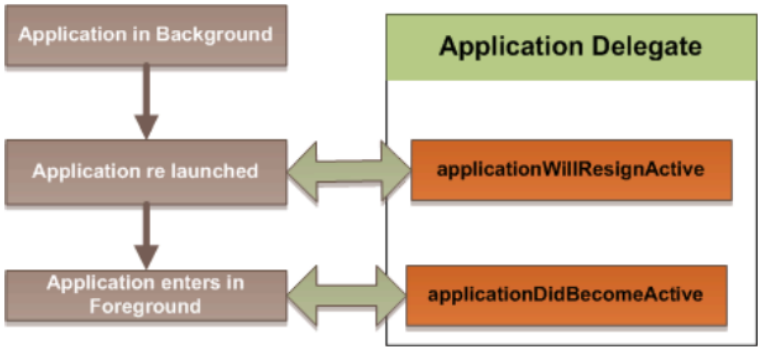


Hình 2.1: Kiến trúc phân tầng IOS

IOS có kiến trúc nhiều tầng, mỗi tầng cung cấp các framework và dịch vụ khác nhau:
Cocoa Touch: Tầng cao nhất, cung cấp cáp framework cốt lõi như UIKit và SwiftUI cho phát triển giao diện người dùng.
Media: Chứa các công nghệ đồ họa, âm thanh và video như Core Graphics, Core Animation, AVFoundation.
Core Services: Cung cấp các dịch vụ cơ bản như Core Data, Core Location, và Foundation framework.
Core OS: Tầng thấp nhất, bao gồm kernel, file system, bảo mật, và các dịch vụ hệ thống cấp thấp.

1.2 Vòng đời ứng dụng IOS

Khi user mở điện thoại của mình lên thì không có ứng dụng nào chạy cả ngoài những thứ nằm trong Operation system . Application của bạn cũng sẽ không chạy. Sau khi user nhấn vào icon của app, Springboard sẽ kích hoạt application của bạn. Application cùng với các thư viện của nó sẽ được thực thi và được tải vào bộ nhớ, trong khi đó thì Springboard sẽ nhận nhiệm vụ hiển thị màn hình launch screen của ứng dụng. Sau cùng thì ứng dụng của bạn bắt đầu được chạy và application delegate sẽ nhận được các notification. Các iOS app chạy trên các thiết bị đều có các trạng thái chuyển đổi như: Not running, In active, Active, Background, Suspended. Tại bất kì thời điểm nào, app của bạn đều rơi vào các trạng thái trên.



Hình 2.2: Vòng đời ứng dụng IOS

NotRunning:Ứng dụng chưa được khởi chạy hoặc đã bị hệ thống chấm dứt.

Inactive: Ứng dụng đang chạy ở foreground nhưng không nhận events (ví dụ: khi có cuộc gọi đến).
Active: Trạng thái bình thường khi ứng dụng chạy ở foreground và đang xử lý events.
Background: Ứng dụng không hiển thị nhưng vẫn chạy và thực thi mã.
Suspended: Ứng dụng ở background nhưng không chạy mã, có thể bị hệ thống chấm dứt để giải phóng tài nguyên.

1.3 App Delegate và Scene Delegate

AppDelegate: Quản lý vòng đời chung của ứng dụng và cấu hình ban đầu.
SceneDelegate: Quản lý UI lifecycle của từng cửa sổ ứng dụng (scene).

1.4 Luồng làm việc của người dùng và Storyboards

Storyboards: Công cụ trực quan để thiết kế và quản lý luồng màn hình trong ứng dụng.
Segues: Xác định chuyển tiếp giữa các màn hình.
Programmatic UI: Phương pháp thay thế để tạo UI bằng mã Swift.

2 Mô hình kiến trúc ứng dụng iOS

Lựa chọn mô hình kiến trúc phù hợp là quyết định quan trọng ảnh hưởng đến khả năng bảo trì, mở rộng và kiểm thử của ứng dụng. Dưới đây là các mô hình phổ biến trong phát triển iOS:

Hình 2.3: Một số mô hình kiến trúc IOS

2.1 Model-View-Controller(MVC)

MVC (Model-View-Controller) là một trong những mô hình kiến trúc truyền thống được Apple khuyến nghị sử dụng trong quá khứ để xây dựng ứng dụng iOS.
Thành phần:

Model: Quản lý dữ liệu và logic nghiệp vụ.

View: Hiển thị giao diện người dùng và phản hồi tương tác.

Controller: Trung gian giữa Model và View, xử lý logic điều khiển.

Ưu điểm:

- Là một trong những mẫu kiến trúc ứng dụng iOS phổ biến nhất, được sử dụng trong cả phát triển ứng dụng và web.

- Dễ dàng phân tách trách nhiệm giữa máy chủ và máy khách.
- Phù hợp cho các ứng dụng nhỏ và đơn giản.

Nhược điểm:

- "Massive View Controller" – Controller thường trở nên quá lớn và phức tạp.
- Khó khăn trong việc kiểm thử.
- Sự phụ thuộc chặt chẽ giữa các thành phần.

2.2 Mô hình MVP trong iOS

MVP (Model-View-Presenter) là một biến thể của MVC, tập trung vào việc tách riêng phần hiển thị (View) và logic điều khiển (Presenter), giúp tăng khả năng kiểm thử và tái sử dụng.

Thành phần:

Model: Tương tự như trong MVC.

View: Giao diện thụ động, chỉ hiển thị dữ liệu.

Presenter: Xử lý logic nghiệp vụ và cập nhật View.

Ưu điểm:

- View hoàn toàn thụ động, dễ kiểm thử.
- Xác minh chức năng chính xác của từng thành phần trở nên dễ tiếp cận hơn trong MVP.
- Presenter có thể được tái sử dụng với các View khác nhau.

Nhược điểm:

- Cần nhiều mã *boilerplate*.
- Presenter có thể trở nên lớn và phức tạp.
- Không phổ biến bằng MVVM trong cộng đồng iOS.

2.3 Mô hình MVVM trong iOS

MVVM (Model-View-ViewModel) là mô hình kiến trúc hiện đại được sử dụng phổ biến trong phát triển ứng dụng iOS, đặc biệt khi kết hợp với các framework reactive như Combine hoặc RxSwift.

Thành phần:

Model: Tương tự như trong MVC.

View: Bao gồm **UIView** và **UIViewController**.

ViewModel: Chuẩn bị dữ liệu từ Model để View hiển thị và xử lý logic.

Bốn nguyên tắc trong MVVM:

The Simplicity Principle (Nguyên tắc đơn giản): Mỗi View chỉ nên có một ViewModel tương ứng. Mỗi ViewModel chỉ phục vụ một View duy nhất.

The Blendability Principle (Nguyên tắc hòa trộn): ViewModel cần hỗ trợ khả năng hòa trộn biểu thức để tối ưu hóa UI.

The Designability Principle (Nguyên tắc thiết kế): ViewModel phải cung cấp dữ liệu có thể dùng tại thời điểm thiết kế (design-time).

The Testability Principle (Nguyên tắc kiểm thử): Cả Model và ViewModel đều phải có khả năng kiểm thử độc lập.

Ưu điểm:

- Tách biệt rõ ràng các thành phần.
- Dễ dàng kiểm thử (đặc biệt là ViewModel).
- Giảm kích thước và trách nhiệm của ViewController.
- Hỗ trợ binding dữ liệu giữa View và ViewModel.

Nhược điểm:

- Phức tạp hơn so với MVC.
- Có thể dẫn đến "Massive ViewModel" nếu không được tổ chức tốt.
- Yêu cầu cơ chế binding (thủ công hoặc sử dụng thư viện reactive).

2.4 MVCVS (Model-View-Controller-ViewState)

MVCVS bằng cách nào đó là sự kết hợp giữa hai kiến trúc MVC và MVVM. Nó giúp tách biệt rõ ràng giữa **Model**, **View**, **Controller**, và **View State**.

Các giai đoạn hoạt động của MVCVS:

MVCVS Initialization: Ở giai đoạn này, View Controller phải tuân theo Model và View State.

MVCVS Model Updates: Nếu có bất kỳ thay đổi nào xảy ra, View Controller sẽ cập nhật Document Model và View State.

MVCVS View Changes: View State phân tích Document View Model và View State. Sau đó, nó thực hiện các thay đổi đối với View dựa trên các quan sát.

MVCVS View State: View Controller và View State được tách biệt. View State chịu trách nhiệm cập nhật và lắng nghe các thay đổi trong View.

MVCVS Testability: Có thể kiểm tra logic của View Model và Document Model một cách riêng biệt, giúp việc kiểm thử hiệu quả hơn so với mô hình MVC.

Ưu điểm:

- Là mẫu kiến trúc có khả năng quản lý và hiệu quả cao.
- Dễ dàng kiểm tra riêng biệt các thành phần nhờ các bài kiểm tra tích hợp.

Nhược điểm:

- Độ phức tạp cao.
- Khó tiếp cận đối với người mới.

2.5 VIPER

Là một trong những mẫu kiến trúc iOS được biết đến với cấu trúc clean. Đây là lựa chọn phù hợp khi cần tạo các thành phần xoay quanh các trường hợp sử dụng cụ thể.

Các thành phần trong VIPER:

View: Giao diện người dùng.

Interactor: Xử lý logic nghiệp vụ.

Presenter: Điều phối giữa View và Interactor.

Entity: Mô hình dữ liệu.

Routing: Điều hướng giữa các màn hình.

Ưu điểm:

- Kiến trúc dễ quản lý, phù hợp với nhóm phát triển lớn.
- Tăng khả năng tái sử dụng mã nguồn.
- UI được xác định rõ ràng và dễ kiểm soát.
- Mã được phân tách giúp kiểm thử dễ dàng.
- Giảm thiểu xung đột hợp nhất mã.

Nhược điểm:

- Cấu trúc phức tạp đối với ứng dụng nhỏ.
- Tốn thời gian để làm quen.
- Nhiều lớp trung gian có thể làm giảm hiệu suất.

2.6 The Elm Architecture (TEA)

TEA là một mô hình kiến trúc mới trong iOS, khá khác biệt so với các cấu trúc Model-X truyền thống. Trạng thái giao diện và mô hình được hợp nhất thành một thực thể duy nhất. Mọi cập nhật được gửi đến thực thể này dưới dạng **messages** và xử lý thông qua **reducers**.

Dòng sự kiện trong TEA là một chiều (unidirectional), tương tự như Flux hoặc Redux.

Ưu điểm:

- View có thể được mô tả như các hàm thuần túy (pure functions).
- One-way binding từ Model đến View giúp dễ kiểm soát.

Nhược điểm:

- Tăng độ phức tạp trong các ứng dụng lớn.
- Không phù hợp với mọi loại ứng dụng.

2.7 Coordinator Pattern

Mặc dù không phải là một mẫu kiến trúc chính thức, Coordinator là một **design pattern** giúp giải quyết các vấn đề điều hướng mà MVC hoặc MVVM không xử lý tốt.

Coordinator chịu trách nhiệm tạo và giữ tham chiếu đến ViewController hiện tại. Nó thực hiện việc điều hướng, ví dụ: hiển thị màn hình mới hoặc đẩy ViewController vào Navigation Controller.

Ưu điểm:

- Tách riêng logic điều hướng, giúp mã dễ bảo trì.
- Tăng tính linh hoạt trong chuyển đổi giữa các màn hình.

Nhược điểm:

- Làm tăng độ phức tạp khi triển khai.
- Cần thay đổi cách tiếp cận luồng dữ liệu.
- Không cần thiết cho các ứng dụng nhỏ.

3 Quản lý trạng thái trong ứng dụng iOS

Quản lý trạng thái hiệu quả là một phần quan trọng trong kiến trúc ứng dụng iOS, đặc biệt khi ứng dụng phát triển về quy mô và độ phức tạp.

3.1 Các cách tiếp cận quản lý trạng thái

1. Quản lý trạng thái cục bộ

- **Trạng thái trong ViewController:** Lưu trữ trạng thái trong các thuộc tính của ViewController.
- **Property Observers:** Sử dụng `didSet` để phản ứng với thay đổi trạng thái.

2. Reactive Programming

- **Combine Framework:** Framework reactive chính thức của Apple.
- **RxSwift:** Thư viện reactive phổ biến trong cộng đồng iOS.

3. Redux-like Architecture

- **Store:** Lưu trữ trạng thái toàn cục.
- **Actions:** Mô tả ý định thay đổi trạng thái.
- **Reducers:** Xử lý các Action và trả về trạng thái mới.

3.2 State Containers

The Composable Architecture (TCA) là một framework được phát triển bởi Point-Free, cung cấp cách tiếp cận có nguyên tắc để xây dựng ứng dụng iOS:

- **State:** Mô tả trạng thái của một feature.
- **Action:** Các sự kiện có thể xảy ra trong ứng dụng.
- **Environment:** Các dependency cần thiết cho logic.
- **Reducer:** Xử lý logic và cập nhật trạng thái tương ứng.

4 Quản lý phụ thuộc

Quản lý phụ thuộc (Dependency Injection - DI) là một kỹ thuật quan trọng để tạo ra mã có thể kiểm thử và bảo trì.

4.1 Các loại Dependency Injection

- **Constructor Injection:** Cung cấp dependencies thông qua initializer.
- **Property Injection:** Cung cấp dependencies thông qua thuộc tính.
- **Method Injection:** Cung cấp dependencies cho một phương thức cụ thể.

4.2 DI Containers

DI Containers giúp quản lý và cung cấp dependencies một cách tự động:

- **Swinject:** Là một DI container phổ biến cho Swift.
- **Service Locator:** Là một giải pháp thay thế cho DI.

5 Cách thiết kế giao diện người dùng

iOS cung cấp hai framework chính để xây dựng giao diện người dùng: **UIKit** và **SwiftUI**.

5.1 UIKit

UIKit là framework UI truyền thống cho iOS, sử dụng mô hình lập trình mệnh lệnh và dựa trên lớp:

- **Auto Layout:** Giúp tạo UI thích ứng với các kích thước màn hình khác nhau.
- **View Controller Lifecycle:** Hiểu rõ vòng đời của **UIViewController** là chìa khóa để quản lý tài nguyên và trạng thái.
- **Reusable Views:** Tái sử dụng views để cải thiện hiệu suất vào khả năng bảo trì.

5.2 SwiftUI

SwiftUI là framework UI khai báo mới của Apple, giới thiệu từ iOS 13.

- **View Basics:** Sử dụng cấu trúc `protocol View` để xây dựng UI.
- **State và Binding:** Sử dụng `property wrappers` để quản lý trạng thái.
- **ObservableObject và EnvironmentObject:** Cung cấp các cơ chế để quản lý trạng thái phức tạp.

5.3 So sánh UIKit và SwiftUI

Khía cạnh	**UIKit**	**SwiftUI**
Năm ra mắt	2008	2019
Loại lập trình	Mệnh lệnh	Khai báo
Cấu trúc	Dựa trên lớp (Class-based)	Dựa trên struct (Struct-based)
Trạng thái	Tự quản lý	Property wrappers
Hỗ trợ iOS	iOS 2+	iOS 13+
Độ ổn định	Rất ổn định	Đang phát triển
Học tập	Phức tạp	Dễ dàng hơn
Tùy biến	Rất linh hoạt	Hạn chế hơn

6 Xử lý Dữ liệu

Quản lý dữ liệu là một phần quan trọng trong phát triển ứng dụng iOS. Apple cung cấp nhiều giải pháp khác nhau để lưu trữ và truy xuất dữ liệu, phù hợp với các mục đích và quy mô sử dụng khác nhau. Dưới đây là so sánh và mô tả chi tiết các giải pháp lưu trữ dữ liệu phổ biến trong hệ sinh thái iOS.

6.1 Core Data

Định nghĩa: Core Data là một framework của Apple dùng để quản lý graph đối tượng và vòng đời dữ liệu trong ứng dụng iOS, không phải là một cơ sở dữ liệu thuần túy.

Kiến trúc Core Data:

- **NSManagedObjectModel:** Mô tả cấu trúc của các entities, attributes và relationships.
- **NSManagedObjectContext:** Không gian làm việc để thực hiện các thao tác CRUD.
- **NSPersistentStoreCoordinator:** Điều phối giữa object model và persistent store.
- **NSPersistentContainer:** Đóng gói stack Core Data (từ iOS 10 trở lên).
- **NSFetchRequest:** Truy vấn dữ liệu từ persistent store.

Đặc điểm chính:

- Quản lý object graph và mối quan hệ phức tạp.
- Lưu trữ dữ liệu sử dụng SQLite backend.
- Tải dữ liệu một cách lười biếng (lazy loading).
- Tích hợp kiểm tra dữ liệu và hỗ trợ migration/versioning.

Ưu điểm:

- Tích hợp tốt với hệ sinh thái Apple.
- Công cụ thiết kế mô hình dữ liệu trực quan.

– Hỗ trợ đồng bộ hóa với CloudKit.

– Quản lý bộ nhớ thông minh.

Nhược điểm:

– Đường cong học tập dốc.

– Debug khó khăn.

– Threading phức tạp và không thread-safe.

Khi nên dùng:

– Ứng dụng có dữ liệu phức tạp với nhiều quan hệ.

– Dự án dài hạn cần bảo trì tốt.

6.2 Realm

Định nghĩa: Realm là cơ sở dữ liệu di động mã nguồn mở, được thiết kế để thay thế Core Data với hiệu suất cao và API đơn giản.

Đặc điểm chính:

– Hoạt động đa nền tảng (iOS, Android).

– Truy cập dữ liệu trực tiếp (zero-copy architecture).

– Hỗ trợ lập trình reactive.

– Thread-safe theo từng instance Realm.

– Hỗ trợ mã hóa và đồng bộ hóa thời gian thực.

Ưu điểm:

– API đơn giản, dễ học.

– Hiệu suất cao.

– Hỗ trợ đồng bộ thời gian thực.

– Tài liệu rõ ràng, cộng đồng lớn.

Nhược điểm:

- Không tích hợp sâu với iOS.
- Không hỗ trợ struct, phải dùng class kế thừa từ `Object`.
- Một số giới hạn trong model threading và truy vấn phức tạp.

Khi nên dùng:

- Ứng dụng cross-platform.
- Dự án yêu cầu hiệu suất cao và real-time sync.
- Ưu tiên offline-first.

6.3 SQLite

Định nghĩa: SQLite là hệ quản trị cơ sở dữ liệu quan hệ nhỏ gọn, không yêu cầu server, được sử dụng phổ biến nhất trên thế giới.

Đặc điểm chính:

- Tự chứa và không cần cấu hình.
- Nhẹ, đáng tin cậy, cross-platform.
- Hỗ trợ đầy đủ SQL.
- Tuân thủ chuẩn ACID.

Ưu điểm:

- Kiểm soát toàn diện schema và truy vấn.
- Phù hợp với các truy vấn phức tạp.
- Dễ dàng backup và khôi phục.

Nhược điểm:

- Phải tự xử lý thread-safe và migration.
- Không có hỗ trợ object mapping tự động.

– Không phù hợp với reactive programming.

Khi nên dùng:

– Cần kiểm soát hoàn toàn cơ sở dữ liệu.

– Ứng dụng cần truy vấn SQL phức tạp.

– Không cần tích hợp chặt với hệ sinh thái iOS.

6.4 UserDefaults

Định nghĩa: Là hệ thống lưu trữ key-value đơn giản, phù hợp cho dữ liệu nhỏ và không nhạy cảm.

Đặc điểm chính:

– Lưu dữ liệu dạng key-value.

– API đơn giản, hỗ trợ các kiểu cơ bản như String, Int, Bool, Date, v.v.

– Tự động đồng bộ hóa và cache nội bộ.

– Có thể đồng bộ với iCloud.

Ưu điểm:

– Dễ sử dụng, hiệu suất cao cho dữ liệu nhỏ.

– Không yêu cầu cấu hình.

– Phù hợp lưu trạng thái người dùng và cài đặt.

Nhược điểm:

– Không phù hợp với dữ liệu lớn hoặc nhạy cảm.

– Không có khả năng filter, query, transaction.

Khi nên dùng:

– Lưu theme, ngôn ngữ, âm thanh.

– Đánh dấu đã hoàn thành hướng dẫn sử dụng.

– Lưu cấu hình đơn giản hoặc flag.

6.5 Keychain

Định nghĩa: Là hệ thống lưu trữ an toàn để bảo vệ thông tin nhạy cảm như mật khẩu và token.

Đặc điểm chính:

- Lưu trữ bảo mật với mã hóa mạnh.
- Tồn tại sau khi ứng dụng bị xóa.
- Hỗ trợ xác thực sinh trắc học.
- Có thể chia sẻ giữa các ứng dụng cùng nhà phát triển.
- Hỗ trợ đồng bộ iCloud.

Ưu điểm:

- Bảo mật cao, phù hợp với dữ liệu nhạy cảm.
- Hỗ trợ xác thực bằng Face ID, Touch ID.
- Dữ liệu tồn tại lâu dài.

Nhược điểm:

- API phức tạp, khó sử dụng hơn so với các giải pháp khác.
- Truy cập và thao tác dữ liệu chậm hơn so với lưu trữ thông thường.
- Khó debug và kiểm tra.

Khi nên dùng:

- Lưu trữ mật khẩu, token xác thực, hoặc thông tin nhạy cảm.
- Cần đảm bảo an toàn dữ liệu người dùng.
- Ứng dụng có yêu cầu bảo mật cao.

7 Xử lý bất đồng bộ

Xử lý tác vụ bất đồng bộ là một phần thiết yếu trong kiến trúc ứng dụng iOS hiện đại, giúp đảm bảo giao diện người dùng mượt mà và phản hồi tốt. Swift cung cấp nhiều phương pháp để xử lý bất đồng bộ, từ truyền thống đến hiện đại, mỗi phương pháp có ưu và nhược điểm riêng phù hợp với từng hoàn cảnh.

7.1 Completion Handlers

Định nghĩa: Completion handlers là cách truyền thống nhất để xử lý bất đồng bộ trong iOS, sử dụng closures được gọi sau khi tác vụ hoàn thành.

Cơ chế hoạt động:

- Hàm bất đồng bộ nhận một closure làm tham số.
- Closure này được gọi khi tác vụ bất đồng bộ hoàn thành.
- Thường sử dụng kiểu `Result` để phân biệt giữa thành công và thất bại.

Ưu điểm:

- Dễ hiểu và được sử dụng rộng rãi.
- Không cần thư viện bên ngoài.
- Tương thích với tất cả các phiên bản iOS.

Nhược điểm:

- Dễ rơi vào *callback hell* khi chuỗi nhiều tác vụ.
- Khó khăn trong việc truyền lỗi giữa các callback.
- Có thể quên gọi completion handler nếu không cẩn thận.
- Khó thực hiện song song hoặc tuần tự tác vụ một cách tối ưu.

7.2 Promises

Định nghĩa: Promises là mô hình xử lý bất đồng bộ dựa trên hướng đối tượng, thường được sử dụng qua các thư viện như `PromiseKit`.

Cơ chế hoạt động:

- `Promise` đại diện cho kết quả trong tương lai.
- Sử dụng `.then` để xử lý chuỗi kết quả.
- `.catch` để xử lý lỗi một cách tập trung.
- Có thể kết hợp và biến đổi Promise dễ dàng.

Ưu điểm:

- Cú pháp rõ ràng, tránh callback hell.
- Dễ dàng thực hiện các tác vụ song song hoặc tuần tự.
- Quản lý lỗi tập trung.
- Hỗ trợ chaining và transform dữ liệu.

Nhược điểm:

- Cần cài đặt thư viện bên thứ ba.
- Có thể mất thời gian làm quen ban đầu.
- Khó debug hơn so với code đồng bộ.

7.3 Async/Await

Định nghĩa: Từ iOS 15, Swift hỗ trợ cú pháp `async/await` để viết code bất đồng bộ như code đồng bộ.

Cơ chế hoạt động:

- `async` đánh dấu hàm bất đồng bộ.
- `await` tạm dừng thực thi cho đến khi có kết quả.
- Hỗ trợ cấu trúc concurrency rõ ràng với `Task`, `TaskGroup`.
- Dễ dàng kết hợp với `try-catch` để xử lý lỗi.

Ưu điểm:

- Cú pháp gọn gàng, dễ đọc.
- Dễ quản lý luồng logic và xử lý lỗi tự nhiên.
- Không cần lồng closure.
- Tích hợp sâu trong hệ sinh thái Apple.

Nhược điểm:

- Yêu cầu iOS 15 trở lên.
- Cần refactor lại call stack cho async.
- Có thể xảy ra rò rỉ bộ nhớ nếu không quản lý `Task.cancel()` cẩn thận.

7.4 Combine Framework

Định nghĩa: Combine là framework lập trình reactive do Apple phát triển, ra mắt từ iOS 13. Thích hợp cho xử lý bất đồng bộ dựa trên dữ liệu và sự kiện.

Cơ chế hoạt động:

- Dựa trên mô hình *Publisher/Subscriber*.
- Tạo chuỗi pipeline để xử lý và biến đổi dữ liệu.
- Hỗ trợ quản lý backpressure.
- Kết hợp mạnh với `SwiftUI`.

Ưu điểm:

- Tích hợp chặt chẽ với Swift và hệ sinh thái Apple.
- Hàng trăm operators hỗ trợ biến đổi và lọc dữ liệu.
- Tự động quản lý bộ nhớ bằng `AnyCancellable`.
- Xử lý đồng bộ và bất đồng bộ theo mô hình dữ liệu.

Nhược điểm:

- Learning curve cao, cần hiểu lập trình reactive.
- Yêu cầu iOS 13 trở lên.
- Debugging các pipeline phức tạp.
- Không hỗ trợ đa nền tảng như `RxSwift`.

8 Kết luận

Kiến trúc ứng dụng iOS hiện đại không chỉ tập trung vào việc phân tách rõ ràng giữa các tầng chức năng như UI, Business Logic và Data, mà còn chú trọng đến khả năng mở rộng, dễ bảo trì và hiệu suất của ứng dụng. Qua việc áp dụng các mô hình kiến trúc như MVC, MVVM, VIPER, kết hợp cùng các kỹ thuật xử lý bất đồng bộ như `Completion Handlers`, `Promises`, `Async/Await`, và `Combine`, các nhà phát triển có thể xây dựng những ứng dụng mượt mà, linh hoạt và thân thiện với người dùng.

Từng cách tiếp cận đều có ưu và nhược điểm riêng, đòi hỏi nhà phát triển cần lựa chọn phù hợp theo yêu cầu của dự án và phiên bản iOS hỗ trợ. Trong đó:

- **Completion Handlers** vẫn là lựa chọn đơn giản và phổ biến.
- **Promises** mang lại cú pháp rõ ràng hơn cho các chuỗi bất đồng bộ.
- **Async/Await** đang dần trở thành tiêu chuẩn mới nhờ vào sự rõ ràng và tích hợp sâu trong ngôn ngữ Swift.
- **Combine** mở ra hướng lập trình phản ứng hiện đại, thích hợp cho các ứng dụng nhiều tương tác và dữ liệu động.

Việc lựa chọn kiến trúc và công nghệ phù hợp không chỉ giúp tối ưu hóa hiệu suất mà còn tạo ra trải nghiệm người dùng vượt trội. Trong bối cảnh hệ sinh thái Apple không ngừng phát triển, việc nắm vững và ứng dụng linh hoạt các kỹ thuật hiện đại sẽ là chìa khóa giúp các nhà phát triển iOS tạo ra những sản phẩm thành công và bền vững.