บทที่ 10

แพลตฟอร์มทางเลือก

อ.ดร.พงษ์ดนัย จิตตวิสุทธิกุล

Topic

3 หัวข้อหลัก:

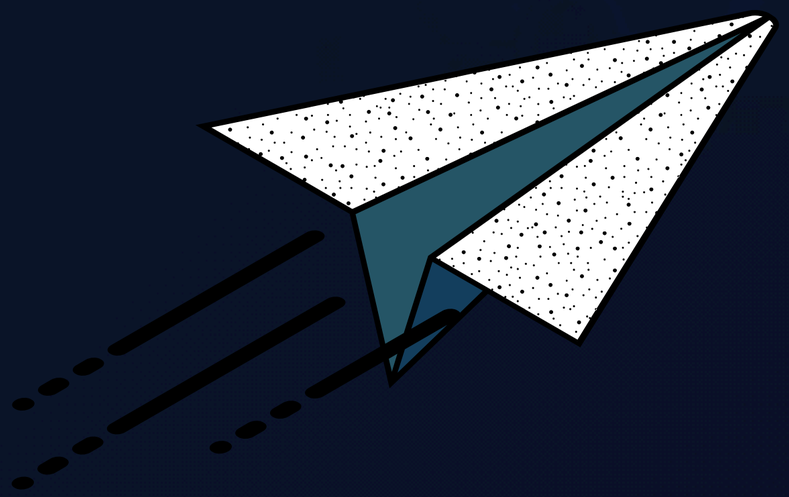
ภาพรวมของการพัฒนา iOS และ Cross-Platform Frameworks

Kotlin Multiplatform

การเลือกเทคโนโลยีในโลกแห่งความเป็นจริง

Objective behavior

3 วัตถุประสงค์เชิงพฤติกรรม:



เข้าใจความแตกต่างระหว่าง Native (iOS), Cross-Platform (Flutter/React Native) และ Shared Logic (KMP)

เข้าใจความแตกต่างของ Rendering Engine (Flutter) กับ JavaScript Bridge (React Native) ที่มีผลต่อประสิทธิภาพ

เลือกเทคโนโลยีที่เหมาะสมกับสถานการณ์สมมติของโปรเจกต์ โดยประเมินจากต้นทุน ประสิทธิภาพ และ UX

นอกเหนือจาก Android Native

โลกของการพัฒนาแอพมือถือไม่ได้มีแค่ระบบเดียว
การรู้กว้างช่วยให้เราตัดสินใจเลือกเครื่องมือที่
เหมาะสมที่สุดได้

การพัฒนาบน iOS (Apple Ecosystem)

ภาษา Swift

ทันสมัย ปลอดภัย และทรงพลัง (เปิดตัวปี 2014 แทน Objective-C)

- **ความปลอดภัย:** ใช้ระบบ ARC (Automatic Reference Counting) จัดการหน่วยความจำ และมีระบบ Optionals จัดการค่า Null
- **ความเร็ว:** ใช้คอมไพเลอร์ LLVM ได้โค้ด Native ประสิทธิภาพสูง

เครื่องมือ Xcode

IDE ศูนย์กลางการพัฒนาครบวงจรของ Apple

- มาพร้อม Interface Builder (ลากวาง UI) และ iOS Simulator

Cross-Platform Frameworks

"เขียนครั้งเดียว รันได้ทุกที่"

ประหยัดเวลา ลดต้นทุน แต่ได้มาซึ่งความท้าทาย

เชิงสถาปัตยกรรม

Flutter: "ศิลปินผู้วาดทุกพิกเซล"

พัฒนาโดย Google (ใช้ภาษา Dart)

Flutter ไม่ได้ใช้ UI Component ดั้งเดิมของ OS แต่ทำงานเหมือน **"ศิลปิน"** ที่ถือผืนผ้าใบและวาดปุ่มขึ้นมาเองทั้งหมด

- **Rendering Engine:** ใช้ Skia (หรือ Impeller) วาดกราฟิก 2D ประสิทธิภาพสูง
- **JIT & AOT:** คอมไพล์แบบ JIT ตอนพัฒนา (ได้ฟีเจอร์ Hot Reload) และคอมไพล์แบบ AOT ตอนปล่อยแอป (ทำงานเร็วระดับ Native)

ผลลัพธ์: UI สอดคล้องกัน 100% ในทุกเครื่อง และ Animation ลื่นไหลมาก

React Native: "สะพานเชื่อมสู่ Native"

พัฒนาโดย Meta (ใช้ JavaScript/React)

React Native ทำงานเหมือน **"สถาปนิก"** ที่สั่งงานช่าง (Native OS) ให้สร้างปุ่มของระบบนั้นๆ ขึ้นมา

- **The Bridge:** ใช้ "สะพาน" สื่อสารระหว่างโค้ด JS (Logic) และ Native Thread (UI) ผ่านข้อความ JSON
- **UI ดั้งเดิม:** แอปจะดูเป็นธรรมชาติ เพราะใช้ Component แท้ของ iOS/Android

จุดอ่อน: Bridge อาจเป็นคอขวด (Bottleneck) ตอนทำ Animation ซับซ้อน (ปัจจุบันแก้ด้วยสถาปัตยกรรม Fabric/JSI)

เปรียบเทียบสถาปัตยกรรม (Flutter vs RN)

คุณสมบัติ	Flutter (Rendering Engine)	React Native (Bridge)
การแสดงผล UI	วาดทุกพิกเซลเองผ่าน Skia/Impeller	แปลงโค้ด JS ไปเรียกใช้ Native UI
ความสอดคล้อง UI	เหมือนกัน 100% ในทุกแพลตฟอร์ม	หน้าตาปรับตามแต่ละ OS โดยอัตโนมัติ
ประสิทธิภาพ	สูงมาก (โดยเฉพาะ Animation)	ดี (แต่อาจมีคอขวดที่ Bridge ในบางกรณี)
ขนาดแอปพลิเคชัน	มักจะใหญ่กว่า (เพราะรวม Engine ไปด้วย)	มักจะเล็กกว่าในตอนเริ่มต้น

Kotlin Multiplatform (KMP): "ทางสายกลาง"

"แชร์เฉพาะสิ่งที่ควรแชร์"

KMP ไม่ใช่การเขียนครั้งเดียวได้ทั้งแอป แต่เป็นการแชร์เฉพาะส่วน ตรรกะทางธุรกิจ (Business Logic) และปล่อยให้ UI เป็น Native 100%

- 📦 **commonMain:** โค้ด Kotlin ส่วนกลาง (Model, API, DB)
- 🤖 **androidMain:** โค้ด Android (สร้าง UI ด้วย Compose)
- 🍎 **iosMain:** โค้ด iOS (ถูกคอมไพล์เป็น Framework นำไปใช้กับ SwiftUI)

กลไก Expect / Actual ใน KMP

หัวใจสำคัญที่ทำให้โค้ดส่วนกลาง สามารถเรียกใช้ฟีเจอร์เฉพาะของ OS (เช่น กล้อง, GPS) ได้

Expect (ความคาดหวัง)

เขียนใน commonMain เพื่อบอกระบบว่า "เราต้องการฟังก์ชันนี้นะ แต่ยังไม่รู้ว่าแต่ละ OS ทำงานยังไง"

```
expect fun getDeviceUUID(): String
```

Actual (การจัดให้)

เขียนใน androidMain / iosMain เพื่อลงรายละเอียดการเขียนโค้ดเรียก Native API ของจริง

```
actual fun getDeviceUUID(): String { ... }
```

ข้อดีและสถานการณ์ที่เหมาะสม

ข้อดีของ KMP

ลดการเขียนโค้ดซ้ำซ้อน ตรรกะทางธุรกิจ (Business Logic) ที่ซับซ้อน จะถูกเขียนและทดสอบเพียงแค่ครั้งเดียว

UX ที่ดีที่สุด (Native UI) แอปมีความลื่นไหลเป็นธรรมชาติ เพราะสร้าง UI ด้วยเครื่องมือแท้ (Compose / SwiftUI)

ประสิทธิภาพระดับ Native โค้ดถูกคอมไพล์เป็น Native Code ของเครื่องนั้นๆ โดยตรง (JVM Bytecode/Native Binary)

นำไปใช้แบบค่อยเป็นค่อยไปได้ สามารถเริ่มใช้กับฟีเจอร์เล็กๆ ในแอปเดิมได้ทันที ไม่จำเป็นต้องรื้อแอปทำใหม่ทั้งหมด

เหมาะกับสถานการณ์ใด?

แอปที่มี Business Logic ซับซ้อน เช่น แอปธนาคาร, แอปคำนวณ หรือแอปจัดการข้อมูลที่ตรรกะการทำงานเหมือนกันทุกแพลตฟอร์ม

ทีมมีความเชี่ยวชาญ Kotlin อยู่แล้ว ทีม Android เดิมสามารถขยายขีดความสามารถไปทำ iOS ได้ โดยยังคงรักษาคุณภาพ UI ระดับสูงสุดไว้

ต้องการคุณภาพสุดยอด แต่คุมต้นทุน โปรเจกต์ที่ต้องการประสิทธิภาพระดับ Native 100% แต่ไม่อยากเสียเงินจ้างคนเขียน Logic ซ้ำสองรอบ

การเลือกเทคโนโลยีในโลกแห่งความเป็นจริง

เกณฑ์พิจารณา	Native (Android/iOS)	Flutter / React Native	Kotlin Multiplatform (KMP)
ประสิทธิภาพ	สูงที่สุด	ดี - สูงมาก	สูงที่สุด (เทียบเท่า Native)
ประสบการณ์ผู้ใช้ (UX)	ดีที่สุด (เป็นธรรมชาติ)	ดีมาก (UI เหมือนกัน / ใช้ Native UI)	ดีที่สุด (UI เป็น Native)
เวลา / ต้นทุนในการพัฒนา	ใช้เวลามาก / ต้นทุนสูง	รวดเร็ว / ต้นทุนต่ำ	ปานกลาง
การบำรุงรักษา (Codebase)	ซับซ้อน (ต้องแก้ 2 ที่)	ง่าย (แก้ 1 ที่)	ปานกลาง (Logic แก้ 1 ที่, UI แก้ 2 ที่)

ปัจจัยในการตัดสินใจ

ปัจจัยอื่น ๆ ที่ต้องพิจารณา

1.ความเชี่ยวชาญของทีม
ทีมมีความถนัดในภาษาและเทคโนโลยีใด

2.ลักษณะของแอปพลิเคชัน
ประสิทธิภาพ ความสามารถและหน้าตาของแอปเป็นแบบใด

3.เป้าหมายทางธุรกิจ
ประสิทธิภาพ ความสามารถและหน้าตาของแอปเป็นแบบใด

Startup == React Native (หรือ Flutter)

แอปธนาคาร == Native แยก 2 แพลตฟอร์ม

Animation แพรวพราว == Flutter

แชร์แค่ตรรกะการคำนวณที่ซับซ้อน == KMP

การอภิปราย

หัวข้ออภิปราย

หัวข้อที่ 1

แนวโน้มในอนาคต Jetpack Compose และ SwiftUI กำลังทำให้การสร้าง UI แบบ Native ง่ายขึ้น สิ่งนี้จะส่งผลกระทบต่อความนิยมของ Cross-Platform Frameworks อย่างไร?

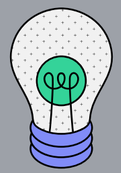
หัวข้อที่ 2

Compose Multiplatform (CMP) ซึ่งเป็นส่วนต่อขยายของ KMP ที่อนุญาตให้แชร์ UI ที่เขียนด้วย Compose ข้ามแพลตฟอร์มได้ จะเข้ามาเป็นคู่แข่งที่น่ากลัวของ Flutter หรือไม่?

หัวข้อที่ 3

ในฐานะนักพัฒนาที่เริ่มต้นจาก Android การเรียนรู้เทคโนโลยีใดเป็นลำดับถัดไปจึงจะคุ้มค่าที่สุดสำหรับสายอาชีพในอีก 5 ปีข้างหน้า?

Questions & Discussion



ไม่มี "กระสุนเงิน" (Silver Bullet) ในการพัฒนาซอฟต์แวร์
การเลือกเทคโนโลยีขึ้นอยู่กับ บริบท ทีม และเป้าหมายทางธุรกิจ