Guitariz 14 минут назад

Заезжаем в Kotlin Multiplatform. Но какой ценой?

14 мин 85

Kotlin*, Разработка мобильных приложений*

Привет! Меня зовут Денис, я занимаюсь коммерческим программированием уже более 15 лет, управляю командами с 2017 года, работал в Яндексе, TradingView, Skillbox, Arcadia. В статье вы узнаете историю, как написать целый проект на Kotlin Multiplatform, сколько это стоит и сколько обойдется интеграция с точки зрения цифр.

Этот доклад - текстовая версия моего выступления с осеннего Мобиуса 2024 года. Обязательно приложу видео с выступлением после выхода.

О чем поговорим в статье:

- Пройдем путь от **идеи** Kotlin Multiplatform до её **реализации** в цифрах.
- Обсудим, сколько стоило внедрить Kotlin Multiplatform в продукты и к чему это привело.

- Проведём ретроспективу: стоило ли это делать вообще и стоит ли инвестировать в эту технологию дальше. А также разберём, сколько это будет стоить в перспективе.

Разбираемся с KMP и CMP: ключевые термины перед стартом

Сначала немного вводных: разберемся с сокращениями, которые часто буду использовать в статье.

KMP (Kotlin Multiplatform) – это технология, которая позволяет шарить код, написанный на Kotlin, между разными платформами (Android, iOS, JVM, Desktop, Web, etc.)

CMP (Compose Multiplatform) — UI, который можно рисовать с помощью Jetpack Compose и портировать на разные устройства. Сейчас официально поддерживаются две платформы: Android и Desktop. iOS находится в стадии публичной беты, а Web – в альфе.

Мы сконцентрируемся на трёх таргетах – Compose Multiplatform для Android, Desktop и iOS. Остальные в статье рассматривать не будем, так как они достаточно редкие или имеют далекий от стабильности статус разработки.

Запуск проекта на KMP + Compose/SwiftUI: архитектура, паттерны и подходы

В апреле 2024 мы начали разработку нового проекта. В качестве базового стека выбрали KMP в связке с Compose Multiplatform и SwiftUI.

Основной подход – использование паттерна MVVM для шаринга кода между экранами разных платформ. В мультиплатформенной части находятся DI, бизнес-логика и переводы, которые поставляются в виде SDK. Каждому экрану доступна своя ViewModel.

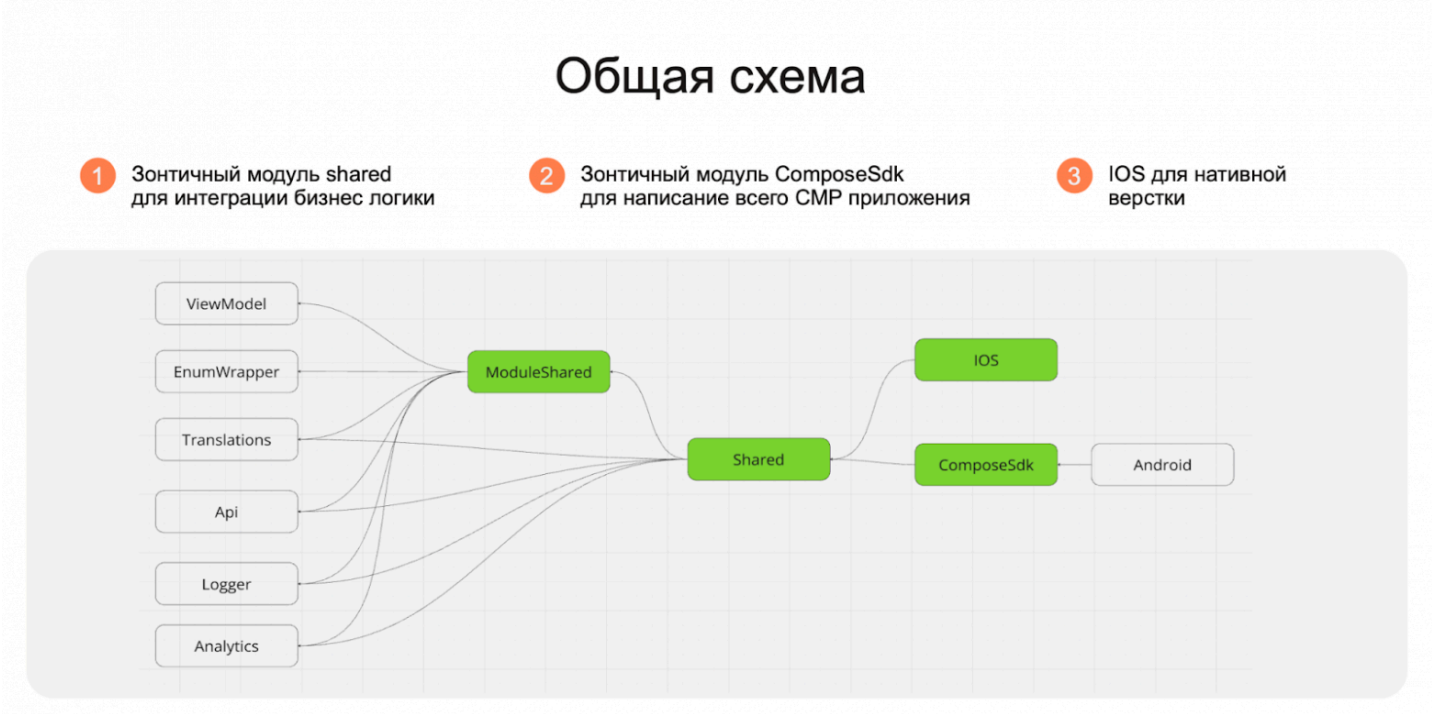
Для Android и iOS мы реализовали следующий подход:

- Android: активити выступает хостом для Composable-функции из библиотеки, а также предоставляет доступ к нативным компонентам.
- iOS: нативный UI на SwiftUI и платформенные связки (например, пуш-уведомления, интеграция с библиотекой авторизации) реализованы отдельно, но используют общее SDK.

Выбор Compose Multiplatform позволил нам легко адаптировать UI для новых таргетов в будущем.

Архитектура проекта: зонтик-модуль, Shared-логика и работа с Compose SDK

Наша схема работы следующая: серыми блоками обозначены элементы, которые мы написали один раз и практически не изменяем, а зелеными — те, к которым мы обращаемся и вносим изменения довольно часто.



У нас есть зонтик-модуль под названием Shared. По сути, это вся логика, написанная на Kotlin Multiplatform, собранная в одном месте, к этому модулю подключаются другие. Например, ModuleShared (какая то отдельная функциональность, например, профиль), который может ссылаться на стандартные решения: API, ViewModel, различные вспомогательные инструменты — переводы, логирование, аналитику и прочее.

Зонтик-модуль мы подключаем к ComposeSdk для того, чтобы получить мультиплатформенный таргет, который хостится в Android-приложении. Compose Sdk также делится на отдельные модули, где имплементированы UI отдельных бизнес фич. А iOS, соответственно, берет просто бизнес-логику Shared целиком и у себя уже имплементирует UI.

Эксперимент 1. Про интеграцию KMP в цифрах

Первое, о чем хочу рассказать, — это сколько стоит интеграция библиотеки Kotlin Multiplatform в мегабайтах и секундах. Мегабайты показывают, насколько ваше приложение увеличится в размере после добавления зависимости. Секунды отражают время, необходимое для горячего перезапуска. Про холодный запуск я тоже упомяну, но в первую очередь нас интересует именно горячий — то, сколько секунд будет уходить на повторную сборку. Это важный параметр, поскольку **он влияет на комфорт работы разработчиков**. Мы будем часто сталкиваться с этим процессом в работе, поэтому важно заранее понять, насколько это критично.

У нас есть таблица с данными. Мы будем постепенно добавлять зависимости в наш проект, тестировать его на трех платформах и сравнивать размеры и время, затрачиваемое на горячую сборку. Давайте начнем с разбора структуры таблицы.

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий

Начнем с чистого проекта. Ну, он не совсем чистый, потому что в Android там добавлен Compose и Kotlin — ведь вы вряд ли будете писать новое Android-приложение без них. То же самое добавлено сразу в JVM Target.

Получаем следующие цифры:

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM

Горячая пересборка (Hot Reload) везде примерно по одной секунде. Размеры нативного iOS-приложения — 0,2 мегабайта (200 килобайт). Android — 5.25, десктоп — 67.85 МБ.

По умолчанию Compose тянет в десктоп-приложение довольно много всего, потому что там внутри JVM (я рассматриваю сейчас таргет под MacOS). Это JVM, которая запускается на Mac OS и тянет за собой немало. Но в целом 67 МБ для нативного приложения, на мой взгляд, — вполне адекватный размер. Предположу, что на этом этапе мы не скажем: "всё, хватит, эксперимент окончен, это слишком много для приложения, и мы не будем его разрабатывать". По моему мнению, для десктопа это разумная цифра для старта.

Теперь посмотрим, как будут расти зависимости.

· **Добавляем Kotlin**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	

Очевидно, он уже есть у нас в Android и также в десктопе. Добавление в iOS весит 2,5 мегабайта. Эти замеры сделаны на версии 2.0.20, но в зависимости от версии языка они могут отличаться.

Я наблюдаю за этим начиная с Kotlin 1.6 — тогда он весил около 1.7 МБ, а сейчас вырос до 2.5, в целом, 2 мегабайта — это все еще немного.

· **Добавляем Coroutines**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	

На Android получается 7,5 мегабайт, а десктоп почему-то не вырос. Там вообще будут разные нюансы, связанные с размером десктопа, — он не всегда растет пропорционально с другими платформами.

Вот такие цифры у меня получились: iOS вырос на 400 килобайт. На мой взгляд, это отличный результат, потому что Coroutines — это мощный инструмент для асинхронной работы, в том числе и в iOS. Под капотом там целый комбайн для работы с асинхронщиной.

· **Добавляем DateTime**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	
DateTime	7.5	1	68.44	1	7.6	1	На холодную 20 сек

Android практически не растет, потому что использует под капотом нативный DateTime. Десктоп незначительно увеличивается на несколько килобайт. А iOS — внимание — 7.6 мегабайта! Почему так? Вероятно, пока библиотека не находится в стабильной версии и “тянет” что-то лишнее. Coroutines добавляют 400 килобайт, а DateTime — 5 мегабайт.

В этот момент горячая сборка всех трёх проектов у меня на M1 занимала примерно 20 секунд с нуля. Повторная сборка — по одной секунде.

· **Добавляем View-Model**

	Android		Desktop		IOS		
							в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	
DateTime	7.5	1	68.44	1	7.6	1	На холодную 20 сек
ViewModel	7.5	1	68.42	1	7.6	1	

В Android она практически ничего не добавляет. Десктоп почему-то стал меньше (помним про возможные ошибки в измерении), iOS тоже не вырос. Были небольшие изменения на уровне байтов и килобайтов, но общий размер в 7.6 мегабайта пока выглядит компактным.

· **Добавляем Ktor**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	
DateTime	7.5	1	68.44	1	7.6	1	На холодную 20 сек
ViewModel	7.5	1	68.42	1	7.6	1	
Ktor	8.63	2	71.37	4	12.9	1	На 2.X было сильно меньше

Это сетевой слой, который умеет ходить в сеть и работает с Coroutines. Android вырастает на 1 мегабайт, десктоп — на 3 мегабайта, а iOS — на 5.3 мегабайта. Здесь я использовал версию Ktor 3.0.0 RC1.

· **Добавляем SqlDelight для базы данных**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	
DateTime	7.5	1	68.44	1	7.6	1	На холодную 20 сек
ViewModel	7.5	1	68.42	1	7.6	1	
Ktor	8.63	2	71.37	4	12.9	1	На 2.X было сильно меньше
SqlDelight	8.7	3	85.07	3	12.9	2	Нативные драйвера

В Android добавляется совсем немного, так как используется нативный драйвер. В iOS прирост минимален по той же причине — применяется нативный SQL-драйвер. А вот в десктопе добавляется заметно больше, поскольку там отсутствует встроенное решение для SQL. Вместо этого подтягивается драйвер JDBC, что добавляет 14 мегабайт. Размер итогового приложения для десктопа — 85 мегабайт, что по-прежнему укладывается в разумные пределы. Мой субъективный психологический барьер в 100 мегабайт пока не достигнут.

· **Добавляем Koin**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	
DateTime	7.5	1	68.44	1	7.6	1	На холодную 20 сек
ViewModel	7.5	1	68.42	1	7.6	1	
Ktor	8.63	2	71.37	4	12.9	1	На 2.X было сильно меньше
SqlDelight	8.7	3	85.07	3	12.9	2	Нативные драйвера
Koin	8.8	1	85.47	2	12.9	2	

Для организации DI-сборки приложение увеличивается на 100 килобайт — как в Android, так и примерно столько же в iOS. В десктопе прирост больше — около 400 килобайт.

- **Compose**

	Android		Desktop		IOS		
Зависимость	Размер	Время	Размер	Время	Размер	Время	Комментарий
Dry	5.25	1	67.84	1	0.2	1	Compose включен в android и JVM
Kotlin	5.25	1	67.84	1	2.7	1	
Coroutines	7.5	1	67.89	1	3.1	1	
DateTime	7.5	1	68.44	1	7.6	1	На холодную 20 сек
ViewModel	7.5	1	68.42	1	7.6	1	
Ktor	8.63	2	71.37	4	12.9	1	На 2.X было сильно меньше
SqlDelight	8.7	3	85.07	3	12.9	2	Нативные драйвера
Koin	8.8	1	85.47	2	12.9	2	
Compose	8.8	1	85.47	3	45	3	

В Android и десктопе не добавляет ничего, потому что он был включен с самого начала. В iOS добавляет 45 мегабайт. Если просто загрузить коробочное решение, то Compose занимает около 30 мегабайт, в наших приложениях это все еще далеко не самая крупная библиотека.

Какие выводы здесь можно сделать?

1. Собирается все очень быстро

Десктоп компилируется примерно **в два раза медленнее**, но это по-прежнему укладывается в рамки секунд. На этапе сборки проблем не возникает.

2. На KMP можно разрабатывать библиотеки

Подключая стандартный набор инструментов Android, мы получаем итоговый вес около 10 мегабайт.

3. Если использовать только стандартный Kotlin, можно уложиться в пару мегабайт

Этот размер включает в себя очень много кода. Если добавить свой код — то прирост будет минимальным и может уложиться всего в пару мегабайт. На практике это практически незаметно.

4. На CMP писать библиотеки сложнее, но тоже можно (около 47 мб)

Считаю, что 47 мегабайт для iOS на сегодняшний день не являются критическим барьером для создания библиотеки. Современные телефоны редко имеют объем памяти меньше 128 гигабайт, так что 47 мегабайт вряд ли создадут проблемы.

Если же разрабатывается простой экран, возможно, стоит задуматься, нужно ли это. Но для более сложных решений, например, платежных систем или систем интеграции заметок, такой подход вполне оправдан.

5. **Десктоп весит и собирается больше остальных, но числа — адекватные**

Он немного тяжелее из-за отсутствия нативного UI и необходимости добавления драйвера JDBC. Однако в целом показатели остаются в пределах нормы.

Эксперимент 2. Сколько стоит написание модуля в байтах

Теперь измерим, сколько добавляет нам написание одного модуля в проекте. Для примера я написал модуль логина. Хотел запустить десктоп с авторизацией — как раз такого модуля для десктопа мне и не хватает, для iOS и Android библиотеки уже есть.

Модуль содержит:

- Три API — ручки на Ktor
- Проперти в SharedPreferences. Замечу, что класть токены туда нельзя, но в данном случае это использовано для примера.
- MVVM и Coroutines под капотом.
- Экран на Compose Multiplatform

Также я продублировал его на SwiftUI для сравнения размеров.

Результаты

	Android	Desktop	iOS2	iOS
Просто зависимости	12.324.456	93.657.439	51.059.43	50.805.815
+1 модуль	12.407.967	93.815.412	51.230.138	57.882.972
+2 экрана	12.443.616	93.839.303	51.232.140	57.883.020

Добавляя зависимости, получаем прирост:

Android — 12 мегабайт.

iOS — 50 мегабайт, второй вариант — 51 мегабайт из-за отличий в версиях библиотек.

Размеры сборок иногда бывают разными, что стоит учитывать. Лично у меня это поведение встречалось, и лучше все перепроверить у себя.

Какие выводы тут можно сделать?

1. Размер приложения растет равномерно и линейно

Исходя из этого, можно долго писать код, не опасаясь, что он превратится в «монстра» на сотни мегабайт.

2. Все приложения растут равномерно (можно ориентироваться на размер Android)

Понимание линейного роста важно для перспективы. Например, если команда из 10 человек разрабатывает приложение в течение двух лет, итоговый размер на iOS можно легко спрогнозировать, добавив небольшую дельту к текущим показателям.

3. Артефакт со сборкой iOS нужно изучить (плавающий инкремент)

4. На время сборки новые модули практически не влияют

5. Модульная архитектура не влияет на размер итогового приложения

> Это подтверждает, что можно создавать гибкие мультимодульные решения, не опасаясь значительного роста веса программы.

Эксперимент 3. Перенос Android Compose на CMP

Теперь посмотрим на то, сколько стоит перенести фичу с Андроида на CMP. Я занимался этим в рамках запуска десктопа — переносил дизайн-систему с Android на Compose Multiplatform.

1) В дизайн-системе у нас было 400 картинок, написанных в формате SVG. Все это мы вытащили в Kotlin-код в Android-библиотеку и прикрутили к Composable теме.

Задача заключалась в том, чтобы из Android-таргета получить мультиплатформенный.

Для этого нам понадобилось

1) Первое изменение — это changelog при перегенерации картинок. Я перегонял их из SVG в код на Kotlin. На это в реальности ушло около 30 минут. Но на код-ревью можно получить 36 тысяч строк changelog'а, просто перенеся картинки из SVG в Kotlin.

2) Я переработал примерно 2000 строк кода — 2000 удалил и 2000 добавил. Суммарно потратил где-то 3 человека-дня.

Я делал это не в первый раз, поэтому без опыта вы потратите немного больше, стоит закладывать в два раза больше времени.

Что было:

· Была довольно понятная логика, ничего такого, что нельзя было бы отревьюить с помощью двух пул-реквестов.

· Один хотфикс — сломались шрифты. Мы использовали Compose 1.6.11. В Compose 1.7.0 этой проблемы уже нет.

· Постоянная, но легкая боль, так как превью в CMP + Android Studio не работает. Решение есть: можно написать превью в Android-части. Оно будет ссылаться на Compose Multiplatform и отображать его, придётся держать два экрана, но это работает.

Нам помогали:

· Самопальный генератор верстки из Figma в Kotlin.

· Плагин для переноса Svg to Compose.

· Бесконечная вера в то, что нам это пригодится.

Мы реально поверили в то, что нам это надо, и сделали. И нам это действительно пригодилось.

Результаты

1. У нас получилась дизайн-система на десктопе.

Можно запускать ее без эмулятора, сравнивать поведение, с помощью мышки, менять размер экрана, смотреть на поведение компонентов и так далее. Это действительно используют сейчас, и мы можем экспериментировать с верстской уже в десктопе.

2. С точки зрения ресурсов — это три человеко-дня.

В дальнейшем поддержка и написание на Compose Multiplatform или Jetpack Compose — для меня и команды особой разницы нет.

3. Дизайнеры благодарны.

Они теперь могут запускать дизайн-систему у себя на макбуке и сразу видеть результат своей работы.

Производительность KMP: тестируем запуск на Android, iOS и Desktop

Теперь мы попробуем измерить загрузку первоначального экрана, запуска кода и оценим, сколько это будет стоить.

Запуск в попугаях

Меряем в "попугаях", потому что реальные цифры будут зависеть от эмулятора и реального устройства — объема памяти, процессора и прочих характеристик. Но даже в таком формате мы сможем получить ориентировочные данные.

1. Запускаем пустой проект с добавленным Kotlin Multiplatform.

2. Запускаем в нем Coroutines. Запустим одну джобу и смотрим, когда она будет выполнена.

3. Затем запускаем то же самое, но с экраном на Compose Multiplatform и фиксируем, сколько миллисекунд уходит с момента запуска приложения до первого показа экрана.

Это не перформанс-ревью, а тестирование того, сколько времени займет старт, и можно ли интегрировать это в приложение так, чтобы процесс оставался комфортным.

Результаты

· Пустой проект в Android запускается за 165 миллисекунд на эмуляторе. Десктоп на старте занял 257 мс, а iOS — внезапно 101 мс. На симуляторе он запускается быстрее всего.

Конечно, iOS здесь немного "читерит", предоставляя симулятор, который фактически работает на macOS без дополнительных прослоек. В то время как десктоп использует JVM, а Android работает через стандартный эмулятор, что влияет на результаты. Числа получены на M1 MacBook с 50 вкладками в браузере и четырьмя мессенджерами — классическая ситуация для разработчика.

· Добавляем вызов корутины.

Число становится чуть меньше. На самом деле Android в начале показывает некоторые колебания, но главное, что разница в конечном счете несущественная.

Десктоп ведет себя так же, потому что там JVM под капотом. Собственно, он дает примерно те же результаты. И внезапно симулятор на iOS выдает меньше времени, чем на старте. Наверное,

потому что я закрыл вкладку с любимым видео у себя на макбуке, поэтому он стал работать немного быстрее.

· Добавляем Compose экран.

Получаем 163 мс в Android, 280 мс в десктопе и 73 мс в симуляторе iOS.

Какие выводы тут можно сделать?

1. Kotlin на iOS не влияет вообще.

А еще работает быстрее, чем в нативном Android. Шутка, потому что размер был в попугаях. Берите сами, замеряйте, сравнивайте, принимайте для себя решения.

2. Compose/Coroutines тоже не влияют на запуск приложения.

Вы видели сами цифры — они никак не повлияли на запуск пустого проекта или последнего со всем стафом внутри.

Эксперимент 4. Запуск Desktop на основе Android

Есть приложение, написанное на Android, и я хочу его запустить у себя на макбуке.

1. Имеем андроид, написанный на технологиях выше.

2. Используем собирающийся десктоп проект.

Он у нас компилировался и собирался для прохождения проверок целостности кода, но при запуске сразу падал с ошибкой в рантайме. В некоторых местах мы использовали expect-actual функции, оставляя заглушки в десктоп-версии. Например, не подключили драйвер для базы данных и не добавили поддержку Ktor. В общем, реальной работоспособности у него не было.

3. Надо прикрутить авторизацию (нет библиотеки).

Мы используем нативные библиотеки авторизации в Android и iOS, а для десктопов такого решения нет (раньше мы разбирали пример написания такой либы).

В итоге на разработку я потратил четыре человека-дня, сделал пять пул-реквестов и долго буксовал, потому что писать свою авторизацию на нативном API — задача, мягко говоря, не самая увлекательная.

Кроме того, кейс сам по себе редкий — в основном используется в ТВ-станциях и подобных устройствах. Например, когда нужно отсканировать QR-код, перейти в браузер для авторизации и затем вернуться в приложение.

Из-за редкости кейса пришлось разбираться и копаться в деталях. В результате проект я запустил за три часа, а на прикручивание авторизации потратил еще три дня.

Какие выводы тут можно сделать?

1. Многие штуки можно переиспользовать с Android.

Например, у нас работает Ktor + OkHTTP, можно взять его прямо из Android и подключить в десктоп.

2. Секьюрных преференсов нет.

Поэтому хранить данные аккаунта и токен там не стоит.

3. Библиотек немного, а те, что есть, не всегда и не во всем поддерживают десктоп.

Но, тем не менее, SqlDelight, Room и Ktor доступны и работают хорошо.

4. UX надо адаптировать, но это решаемая проблема.

Если вы захотите поддержать планшетную верстку, так или иначе придется это сделать.

5. В целом, все работает так же, как и на других платформах.

Compose на десктопе — ничего сверхъестественного.

Экономия времени и ресурсов: плюсы и минусы мультиплатформенного подхода

1. Мы пишем одно ядро приложения один раз.

2. UI пока пишется дважды, но UX адаптирован.

Единая ViewModel позволяет сократить работу. Например, показы тостов и разрешений вызываются одинаково.

3. Преимущество CMP с ненативным UI пока не очевидно ни по скорости запуска, ни по времени работы.

Тем более это все еще бета-версия, поэтому поддержка может быть сложной. JetBrains постепенно подтягивает недостающие компоненты, Material 3 уже есть в Compose 1.7.0.

4. Баги исправляются сразу на двух платформах, что экономит время.

На тестах тоже экономим, потому что бизнес-логику можно проверить на одной платформе, а она будет работать и на второй. Но тестировать UI все равно придется отдельно, так как приложения хоть и похожи, но разные.

5. **Экономия времени**

Я оцениваю экономию в 30%. Мы делим 100% времени пополам на Android и iOS. Android делим на 30% для бизнес-логики и 20% на UI. С iOS забираем те же 30% логики, оставляя 70%.

Экономия — 30%.

Конечно, все зависит от того, как вы пишете код, и сколько времени уходит на бизнес-логику или UI. 50% сэкономить невозможно, так как Android-приложение не запустится на iOS без доработок.

Тестирование в мультиплатформе: где экономим, а где — нет

· Дешевле писать тесты

Тесты на KMP можно запускать везде, сравнивать результаты и хостить.

· Из-за разных UI тестировать их придется отдельно.

Проверять баги на промежуточной платформе тоже необходимо. Если мы не уверены на 100%, что проблема в бизнес-логике, а тестировщики тоже не могут это подтвердить, баг все равно придется проверять на обеих платформах.

· Экономия на половине тестов может составить примерно 10% от общего объема работ по тестированию.

Но это расчет в условных единицах — «в попугаях». Здесь каждый может для себя визуализировать цифры по-своему.

· Багов в целом получается объективно меньше.

Кодовая база сокращается, у нас один источник истины, и изменения вносятся в одном месте. Это снижает количество точек, где могут возникнуть баги.

Конечно, с увеличением числа поддерживаемых платформ нагрузка возрастает, но общая выгода от подхода дает профит.

Сборка мультиплатформенного проекта: ожидания vs реальность

· **Сборка проектов у нас идет параллельно.**

Это важно, так как позволяет проверять работу кода одновременно на Android и iOS. Общее время сборки при этом определяется самым долгим пайплайном.

В нашем случае это обычно iOS, хотя проблема связана не столько с самой платформой, сколько с виртуальными машинами, которые мы используем.

· **Работа с единым источником истины.**
Когда в core-компоненте вносятся изменения, сборка триггерится сразу на обоих таргетах. После этого проверяем, работают ли изменения и подтянулись ли они корректно.

· **Сборка для iOS может быть довольно проблемной.**

Интеграция с KMP в Xcode оставляет желать лучшего (впрочем, вряд ли можно рассчитывать на это со стороны Apple), и добавление зависимостей на этапе pull request может превратиться в квест “угадай, почему xcode это не видит”.

Затаскивание новых файлов в iOS-проект — отдельная боль. Добавьте сюда еще использование сторонних систем сборки, которые только усложняют процесс.

По факту мультиплатформенный подход не всегда упрощает процесс сборки, а часто наоборот — добавляет сложности.

· **Можно начинать разработку с Android.**

Это позволяет не тратить время на постоянные сборки и проверки iOS на ранних этапах. Логику и UI тестируем сначала в Android. После этого переносим в iOS, минимизируя доработки и отладку.

Этот подход экономит время, хотя иногда приходится возвращаться назад для внесения изменений. Android собирается быстрее, так как для него это нативный процесс.

· **Для iOS сборка идет медленнее Android (хотя это все еще измерение в попугаях).**

Если говорить в цифрах:

Android — 1 минута 15 секунд против iOS — 2,5 минуты.
Суммарно весь пайплайн у нас работает примерно в 2 раза дольше для iOS, чем для Android.

Почему с iOS все не просто

В iOS есть свои особенности сборки и дополнительные сложности.

Некоторые из них связаны с интеграцией библиотек и зависимостей, другие — с особенностями взаимодействия с системами сборки и пайплайнами.

Транзитивные зависимости не работают. Нужен модуль-зонтик для подключения и мержа зависимостей. Отсюда все недостатки зонтичной системы наследования.

Все не так драматично (в числах все равно получается довольно быстро). Мы получаем незначительные изменения в разных платформах. Но сам подход написания большого зонтичного модуля, хотелось бы видеть только на этапе сборки конечной программы, а не промежуточного модуля.

Что в итоге

Использование Kotlin Multiplatform в разработке оказалось не таким болезненным, как может показаться на первый взгляд. Общая бизнес-логика действительно позволяет сократить затраты на разработку примерно на 30%, а аналогичную экономию можно получить и на тестировании. При этом использование Jetpack Compose уже сейчас выглядит оправданным шагом, особенно если закладывать возможность перехода на Compose Multiplatform в будущем — это бесплатно и может дать преимущества. Однако полностью переходить на CMP пока рискованно, поскольку