Kotlin Multiplatform

на практике

Анна Жаркова

Анна Жаркова

Kotlin Multiplatform

на практике

ПИТЕР

Санкт-Петербург • Москва • Минск

2026

Анна Жаркова

Kotlin Multiplatform на практике

Серия «Библиотека программиста»

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*Н. Михеева*
Ведущий редактор	*С. Шевякова*
Литературный редактор	*А. Устинова*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*

ББК 32.973.2-018
УДК 004.738.5

Жаркова Анна

Ж35 Kotlin Multiplatform на практике. — СПб.: Питер, 2026. — 384 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4406-8

Хотите писать код один раз и запускать его на Android и iOS, сохранив нативное качество и производительность? В этой книге вы освоите все шаги разработки полнофункциональных кросс-платформенных приложений с помощью Kotlin Multiplatform SDK — от создания проекта с нуля до миграции существующих нативных решений. Подробные инструкции, архитектурные подходы и реальные примеры помогут уверенно работать с общими модулями, многопоточностью, сетевым слоем, хранением данных и Compose Multiplatform для iOS.

Особое внимание уделено использованию нативного и общего кода в кросс-платформенном приложении, применению expect/actual, тестированию с Kotest, оптимизации разработки с помощью KSP, плагинов компилятора и ИИ-ассистентов. Вы узнаете, как создавать и публиковать собственные KMP-библиотеки, а также получите краткий справочник по Kotlin и Swift, чтобы уверенно ориентироваться в обеих экосистемах.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-5-4461-4406-8

© ООО Издательство «Питер», 2025
© Серия «Библиотека программиста», 2025
© Анна Жаркова, 2025

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 15.11.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 30,960. Доп. тираж. Заказ 0000.

Оглавление

От издательства **8**

О книге **9**

Для кого эта книга 9

Структура книги 9

Условные обозначения 11

От автора **12**

ЧАСТЬ I

Вводная глава. Почему Kotlin Multiplatform **14**

Польза кросс-платформенной разработки 14

История Kotlin Multiplatform 14

Почему Kotlin Multiplatform 15

Когда не стоит использовать Kotlin Multiplatform 16

Дорожная карта развития Kotlin Multiplatform 17

Глава 1. Создаем и настраиваем проект Kotlin Multiplatform **18**

1.1. Создание проекта с нуля 22

1.2. Настройка проекта и управление зависимостями 25

1.3. Shared-модуль приложения 30

1.4. Создание проекта из существующего нативного 37

1.5. Пробуем Fleet 39

1.5.1. Kotlin Multiplatform Wizard 40

Глава 2. Подготовка приложения на KMP. Старт разработки **43**

2.1. Каркас приложения 44

2.2. Архитектура кросс-платформенного приложения 49

2.2.1. Нативное приложение и общий интерфейс 49

2.2.2. Бизнес-логика экранного модуля 55

2.2.3. Общий архитектурный посредник 57

2.3. UI нативных приложений 61

2.3.1. UI на SwiftUI 61

2.3.2. UI на Jetpack Compose 64

Глава 3. Expect/actual. Общее и нативное 67
3.1. Интеропедия. Библиотека преобразований Kotlin в Objective-C/Swift.....69
3.1.1. Базовые примеры конвертации 71
3.1.2. Ограничения конвертации Kotlin в Swift 75
3.2. Интерфейс или expect/actual 78
3.2.1. Приложение с нативным UI и общим интерфейсом для View.....78
3.2.2. Интерфейс vs. expect/actual.....84
3.2.3. Платформо-специфичное поведение. Чтение файла.....90
3.2.4. Связь типов.....93
3.3. Материал со звездочкой. Как работает под капотом интероп 94
3.4. Из Kotlin в Swift и обратно 104

Глава 4. Многопоточность в приложениях Kotlin Multiplatform.....109
4.1. Coroutines для iOS-разработчиков 110
4.2. Совсем общая многопоточность. Настройка корутин для iOS и Kotlin/Native 115
4.3. Использование корутин и скоупов..... 119
4.4. Под капотом Dispatchers.IO..... 120
4.5. Материал со звездочкой. Внутренняя кухня многопоточности в Kotlin/Native..... 122
4.6. Управление памятью..... 128
4.7. Соединяем suspend с iOS 129
4.8. Соединяем Kotlin Flow с iOS..... 131
4.9. Применение на практике..... 138

Глава 5. Сетевой слой приложения KMP 140
5.1. Подготовка к работе..... 140
5.2. Нативная реализация в commonMain..... 145
5.3. Общий сетевой клиент на Ktor 151
5.4. Материал со звездочкой. Внутренняя кухня Ktor 156
5.5. Применение на практике..... 160

Глава 6. Собираем приложение по частям 161
6.1. Соединение компонентов решения 161
6.2. Миграция на готовую ViewModel..... 167
6.3. Материал со звездочкой. Что под капотом ViewModel..... 170

Глава 7. Связываем слои. DI в Kotlin Multiplatform 174
7.1. Концепция общего Dependency Injection (DI)..... 174
7.2. Начало работы с Kodein 181
7.3. Работа с Koin DI..... 185

7.4. Синтаксические приемы для оптимизации кода ... 190
7.5. Kotlin Inject ... 192
7.6. Dagger Multiplatform ... 193
7.7. Доработка приложения ... 194

Глава 8. Храним данные в приложениях Kotlin Multiplatform ... 197
8.1. Нативные хранилища iOS и Android ... 197
8.2. DataStore в Kotlin Multiplatform ... 206
8.3. Room ... 208
8.3.1. Установка и подключение Room ... 209
8.3.2. Настройка хранилища Room ... 212
8.3.3. Работа с хранилищем ... 218
8.3.4. Ограничения Room KMP ... 219
8.4. Применение на практике ... 221

Глава 9. Тестируем приложение Kotlin Multiplatform ... 226
9.1. Kotlin Test ... 227
9.2. Работа с моками для комплексных тестов ... 232
9.3. Платформенные тесты ... 239
9.4. Expect/actual в тестах ... 242
9.5. Kotest ... 245
9.6. Применение на практике ... 248

Глава 10. Общий UI в приложениях Kotlin Multiplatform ... 251
10.1. Концепция общего UI ... 251
10.1.1. Нативный UI в iOS ... 251
10.1.2. Нативный UI для Android ... 256
10.2. Общий UI ... 259
10.2.1. Общая обертка для UI ... 261
10.2.2. Общий контрол в нативной обертке ... 262
10.2.3. Подключение через alias ... 264
10.3. Compose Multiplatform ... 265
10.3.1. Подключение и настройка CMP в проекте ... 267
10.3.2. Перенос и адаптация UI под Compose Multiplatform ... 269
10.4. Навигация в проектах с Compose Multiplatform ... 272
10.4.1. Навигация на PreCompose ... 273
10.4.2. Нативная навигация от JetBrains ... 279
10.5. Подключение нативного кода к Compose Multiplatform ... 283
10.6. Server Driven UI ... 286

10.7. Альтернативные решения для общего UI ... 291
10.7.1. Библиотека Redwood ... 291
10.7.2. Библиотека Decompose ... 292

ЧАСТЬ II

Глава 11. Полезные инструменты для Kotlin Multiplatform ... 298
11.1. Аннотации и Kotlin Symbol Processing ... 298
11.2. Плагины компилятора ... 305
11.3. ИИ-помощники для кросс-платформенной разработки ... 310
11.3.1. Gemini ... 311
11.3.2. GigaCode ... 327
11.4. Полезные библиотеки для Kotlin Multiplatform ... 330

Глава 12. iOS для Android и Android для iOS. Шпаргалка по языкам ... 332
12.1. Kotlin для разработчиков iOS и Swift для разработчиков Android ... 332
12.1.1. Переменные и константы ... 332
12.1.2. Опционалы, нуллабельные типы ... 333
12.1.3. Потоки исполнения ... 335
12.1.4. Функции ... 336
12.1.5. Структуры и классы ... 337
12.1.6. Расширения в Kotlin и Swift ... 338
12.1.7. Enum, enum class, sealed class ... 338
12.1.8. Синглтоны и статический функционал ... 339
12.1.9. Интерфейсы и протоколы ... 340
12.2. Gradle для разработчиков iOS ... 341
12.2.1. Settings.gradle ... 342
12.2.2. Build.gradle.kts (project) ... 343
12.2.3. Build.gradle.kts shared-модуля ... 343
12.2.4. Gradle.properties ... 346

Глава 13. Создаем кросс-платформенную библиотеку для приложения Kotlin Multiplatform ... 347
13.1. Создание библиотеки ... 347
13.2. Сетевая библиотека для Kotlin Multiplatform ... 350
13.3. Сборка библиотеки ... 354
13.3.1. Сборка артефактов ... 354
13.3.2. Подключение к iOS-приложению ... 356
13.3.3. Подключение к приложению Android ... 360

13.4. Упаковка и публикация артефакта для Maven 362
13.5. Размещение библиотеки в публичном репозитории 364
13.5.1. Github Packages 365
13.5.2. Публикация в Maven Central 366

Глава 14. Гайд-шпаргалка по миграции существующих нативных приложений 372
14.1. Из Android в Kotlin Multiplatform 372
14.2. Миграция существующего iOS-приложения 376

Резюме 381

Что дальше 383

Перспективы Kotlin Multiplatform 384

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

О книге

В этой книге вы узнаете о SDK Kotlin Multiplatform и разработке полнофункциональных кросс-платформенных приложений. Мы рассмотрим как создание приложения с нуля, так и миграцию существующих нативных приложений на Kotlin Multiplatform (KMP). Книга содержит пошаговую инструкцию по реализации различных функциональных слоев приложения и использованию инструментария для контроля качества (покрытие тестами), а также способы оптимизации и ускорения разработки (KSP, плагины компилятора, помощники ИИ). Затрагиваются вопросы применения технологии Compose Multiplatform iOS. Кроме того, из книги вы узнаете, как создать и опубликовать свою библиотеку для Kotlin Multiplatform. В главе 12 вы найдете краткий справочник базовой информации по языкам Kotlin и Swift.

Для кого эта книга

Книга предназначена для разработчиков Android и/или iOS, имеющих определенный опыт в работе с нативной платформой и желающих освоить кросс-платформенную разработку на Kotlin Multiplatform. Любой кросс-платформенный SDK или фреймворк основывается на нативном коде, поэтому знания по нему, хотя бы базовые, необходимы. Мы затронули особенности работы на базе и Android, и iOS и стремились сделать так, чтобы материал был понятен широкому кругу разработчиков. В процессе чтения вы начнете изучать помимо привычной вам и вторую платформу, поэтому рекомендуем ознакомиться с дополнительной информацией о ней.

Структура книги

Книга состоит из двух частей. Первая включает десять глав. Это своего рода дорожная карта, которая поможет освоить все этапы разработки полнофункционального кросс-платформенного приложения KMP и отработать их на практике.

В главе 1 мы расскажем, как установить и настроить средства разработки приложений с помощью Kotlin Multiplatform. Также рассмотрим особенности структуры базового приложения KMP, в том числе работу с общим модулем при создании кросс-платформенного приложения с нуля или подключении к существующим нативным приложениям.

В главе 2 описывается учебное приложение, которое мы будем писать пошагово, реализуя те или иные части при прохождении определенной главы. Мы познакомимся с возможными архитектурами кросс-платформенного приложения, его каркасом, работой с нативным UI и уровнями разделяемого кода.

Глава 3 посвящена концепции общего и нативного в KMP. В общем модуле приложения есть как полностью общая логика, так и платформо-специфичная. Поговорим о возможных способах реализации подхода `expect/actual` и правильном их использовании в различных случаях. Также подробно разберем, как происходит преобразование Kotlin в Swift/Objective-C под капотом, и изучим нюансы компиляции. Эта часть будет вынесена в разделы, названные «Материал со звездочкой».

В главе 4 обсудим реализацию многопоточности в KMP и управление памятью с погружением в глубины этого механизма. Не переживайте, вы можете прочитать главу до «Материала со звездочкой» и вернуться к нему при необходимости. Дополнительно расскажем про механизм корутин.

В главе 5 переходим к разработке сетевого слоя приложения. Рассмотрим, как реализовать общий сетевой клиент, полностью идентичный нативному для Android и iOS, но средствами и инструментами Kotlin. Также научимся просто и быстро разрабатывать кросс-платформенный сетевой слой на Ktor с учетом всех его особенностей.

В главе 6 резюмируем материал и практические примеры, данные в предыдущих главах. Мы соберем из них учебное приложение и проведем миграцию с нашей самописной архитектуры на готовую мультиплатформенную `ViewModel` от Google.

В главе 7 разберемся, как с помощью механизма Dependency Injection соединить функциональные слои приложения: какие решения существуют, как их правильно использовать.

В главе 8 перечислим способы хранения данных в приложениях Kotlin Multiplatform: локальные нативные хранилища, общее хранилище DataStore, база данных Room для KMP.

В главе 9 обсудим способы проверки качества приложения. Рассмотрим особенности тестирования кросс-платформенного приложения, существующие решения, такие как Kotest и Kotlin Test, и их использование на практике.

В главе 10 рассмотрим создание общего UI на Compose Multiplatform, затронем возможные и существующие подходы к организации взаимодействия между нативным и общим UI, способы реализации навигации и работы с состоянием, а также подходящие библиотеки.

Вторая часть, объединяющая четыре главы, даст дополнительную полезную информацию, которая поможет вам углубить свои знания.

В главе 11 вы найдете инструменты, которые облегчат вашу жизнь в будущем при создании KMP-приложения, узнаете способы оптимизации и ускорения разработки, такие как кодогенерация KSP, плагины компилятора Kotlin и ассистенты ИИ.

В главе 12 приведена шпаргалка по Swift, Kotlin и Gradle. Разработчикам iOS будет интересно получить базовую информацию о Kotlin и Gradle, а разработчикам Android — познакомиться со Swift.

В главе 13 вы узнаете, как создать кросс-платформенную библиотеку, опубликовать ее с помощью Github Actions, а также как собрать и загрузить артефакты Maven.

В главе 14 даем краткое руководство по миграции существующего приложения и его переходу от нативного Android или iOS к кросс-платформенному.

Условные обозначения

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, библиотеки, интерфейсы, пути, имена файлов и их расширений.

Шрифт без засечек

Используется для обозначения элементов интерфейса, папок, каталогов, URL и адресов электронной почты.

```
Листинги
```

Серым цветом выделены фрагменты, где в скрипты добавлялись зависимости.

От автора

Приветствую вас!

Меня зовут Анна Жаркова, я разработчик мобильных приложений с опытом работы более десяти лет. Занимаюсь разработкой нативных iOS- и Android-приложений и популяризацией SDK Kotlin Multiplatform как эксперт Ex-GDE Kotlin. Кроме того, я автор статей по разработке на Habr и Medium и преподаватель на курсах по iOS в Otus. Веду Telegram-канал «Записки разработчицы» (@prettygeeknotes).

С Kotlin Multiplatform я познакомилась в 2020 году, еще до появления первой alpha-версии плагина KMM. И с тех пор я плотно изучаю эту технологию и ее возможности, работаю с ней в личных проектах и ищу наилучшие способы решения самых разных задач. Я неоднократно выступала с докладами на темы, связанные с KMP, на многих мероприятиях: Mobius, Droidcon, Android Worldwide и других.

Теперь я решила поделиться накопленным опытом с вами в этой уникальной книге. Она посвящена разработке кросс-платформенных приложений на Kotlin Multiplatform. В ней вы найдете и полезную информацию по работе с этой технологией, и примеры ее нетривиального использования — всё, что поможет вам обойти подводные камни на пути создания приложения. Максимальную пользу вы получите, прочитав книгу от начала и до конца и отработав практические примеры самостоятельно. Это позволит вам в совершенстве освоить кросс-платформенную разработку. Также вы можете читать эту книгу как справочник: только те главы, которые вам интересны. С ее помощью вы не просто изучите KMP, но и станете более квалифицированным специалистом.

Код для примеров из книги смотрите здесь: https://gitverse.ru/azharkova/kmp_book.

Перейдя по этому QR-коду, вы увидите список полезных ссылок из этой книги. Если у вас возникнет проблема с переходом по QR, то список литературы вы найдете также по ссылке https://gitverse.ru/azharkova/kmp_book.

ЧАСТЬ I

ВВОДНАЯ ГЛАВА

Почему Kotlin Multiplatform

Польза кросс-платформенной разработки

Сколько существует разработка мобильных приложений, столько разработчики ищут способы и варианты ускорить и упростить создание готового продукта.

Несмотря на развитие и совершенствование нативного инструментария для iOS и Android, различные SDK и кросс-платформенные фреймворки по-прежнему остаются привлекательными для разработчиков и компаний. Они имеют немало преимуществ, основное из которых — возможность писать единый код для нескольких платформ, в частности для iOS и Android, одновременно.

Такой подход позволяет значительно сократить время разработки и снизить затраты на поддержку двух отдельных кодовых баз. А благодаря тому что изменения вносятся в общий код, а не дублируются для каждой платформы, приложение проще обновлять и поддерживать. Это особенно важно для стартапов и небольших компаний с ограниченными ресурсами.

Современные решения для кросс-платформенной разработки практически не уступают нативным в производительности. Также у них довольно широкий инструментарий и надежная поддержка готовых решений.

Одним из современных решений кросс-платформенной разработки мобильных приложений является SDK Kotlin Multiplatform (KMP) от компании JetBrains. Согласно концепции авторов, это не фреймворк, так как изначально в нем не было средств разработки общего UI и работы с узкими платформенными зависимостями (сенсорами). Это именно SDK, который позволяет создавать модули с общим кодом, подключаемые к нативным приложениям. Затем эти модули компилируются в формат библиотеки, подходящий для конкретных платформ.

История Kotlin Multiplatform

Впервые SDK Kotlin Multiplatform был представлен на Kotlin Conf 2017 под названием Kotlin Multiplatform Projects. Изначально данная технология была нацелена на поддержку таргетов Android, JS и Native. До 2020 года KMP находился в экспериментальной стадии разработки, которая велась по отзывам первых пользователей и контрибьюторов.

В 2020 году компания переключилась на поддержку и разработку технологии разделения кода между Android и iOS как на самый распространенный вариант использования SDK. Одной из целей компании стала популяризация KMP среди мобильных разработчиков. Все, что связано с совместным использованием кода между iOS и Android, было выделено в отдельный продукт Kotlin Multiplatform Mobile (KMM), который перешел в alpha-стадию разработки. Осенью того же года JetBrains выпустила плагин для Android Studio Kotlin Multiplatform Mobile.

В 2023 году вышел первый стабильный релиз Kotlin Multiplatform. К тому времени компания JetBrains решила не ограничивать возможности технологии только мобильной разработкой. В качестве платформ стали рассматриваться также веб (WASM) и Desktop (Linux, Mac OS, Windows).

Еще одним ключевым направлением развития Kotlin Multiplatform стала разработка кросс-платформенного фреймворка для создания UI в приложениях KMP, Compose Multiplatform.

В том же 2023 году, еще до выхода стабильной версии Kotlin Multiplatform, компания Google объявила о поддержке этой технологии в рамках Google Workspace. При сотрудничестве с компанией JetBrains многие библиотеки из набора решений Jetpack (библиотеки Room, DataStore, ViewModel и другие) для Android были адаптированы для работы с Kotlin Multiplatform, новые решения создаются уже с поддержкой кросс-платформы. Также Google официально рекомендовал использовать KMP как средство разработки эффективной бизнес-логики для Android-приложений (https://android-developers.googleblog.com/2024/05/android-support-for-kotlin-multiplatform-to-share-business-logic-across-mobile-web-server-desktop.html).

Почему Kotlin Multiplatform

Kotlin Multiplatform не единственная технология кросс-платформенной разработки. В качестве альтернативных решений можно назвать Flutter, React Native, гибридные SDK и Xamarin MAUI. Однако у KMP есть ряд преимуществ по сравнению с ними:

- Прямая компиляция. Модуль разделяемого кода компилируется в нативный код для платформы напрямую, без дополнительных VM (например, Dart VM) и преобразований (как JavaScript в React Native)).
- Разработка на Kotlin. Разработчикам Android практически не нужно ничего менять в своем коде, чтобы перенести его в разделяемый модуль, вхождение в новую технологию для них не представляет сложности. Код на Kotlin также будет интуитивно понятен разработчикам Swift. Благодаря документации по соотношению конструкций Kotlin и Swift и этой книге, освоить KMP и начать писать на нем им будет существенно проще, чем изучить с нуля совершенно неизвестные Dart или JavaScript.

- Гибкость использования. Современная реализация Kotlin Multiplatform позволяет использовать его как модуль разделяемой бизнес-логики, встраиваемый в приложения с нативным UI. Также с помощью KMP и Compose Multiplatform можно реализовать приложение, полностью кросс-платформенное на всех уровнях логики.
- Полная интеграция с платформенным API. KMP поддерживает специальный механизм `expect/actual` для вызова платформо-специфичного кода в рамках разделяемого модуля.
- Не только мобильная разработка. Kotlin Multiplatform поддерживает разработку и для десктоп-приложений, и для веб с помощью WASM.
- Google активно поддерживает и развивает Kotlin Multiplatform и рекомендует его к использованию как самостоятельное решение и как инструмент создания бизнес-логики для Android-приложений.
- Изучая Kotlin Multiplatform, вы знакомитесь еще с одной платформой нативной разработки. В процессе работы с этим SDK, решая задачи по мере их поступления, вы постепенно погрузитесь в нюансы реализации второй платформы. Это удобный способ узнать больше об iOS или Android.

Когда не стоит использовать Kotlin Multiplatform

Есть случаи, когда KMP не подойдет для разработки:

1. Нужно адаптировать сложный UI и/или готовые решения на нативных библиотеках под кросс-платформу в ограниченные сроки. Перенос готового сложного нативного решения на кросс-платформу может занимать довольно много времени. Поэтому тут надо исходить из возможностей и ресурсов проекта.
2. Проект с упором на веб. Kotlin/WASM сейчас находится на alpha-стадии разработки. Согласно предупреждению на сайте документации, его API нестабилен и может измениться в любой момент. Поэтому для настоящего продукта эта технология сейчас не подходит.
3. Команда не знает и не хочет учить Kotlin. К сожалению, если команда разработки откажется изучать новую технологию и погружаться в новый для себя стек и не выйдет ее переубедить, то использовать KMP в работе просто не получится. В таком случае имеет смысл рассмотреть альтернативную платформу или возможность писать код нативно.

Если вы готовы учиться и осваивать новую для себя технологию Kotlin Multiplatform или хотите углубить свои знания, то эта книга как раз для вас. А если вы управляете командой разработки, то после прочтения вы точно сможете найти аргументы, почему вам на проекте нужен Kotlin Multiplatform.

Дорожная карта развития Kotlin Multiplatform

Основной вектор развития SDK Kotlin Multiplatform направлен на поддержание новых возможностей языка Kotlin, улучшение пользовательского опыта, повышение производительности SDK и оптимизацию, а также рост и развитие экосистемы KMP.

Ключевыми приоритетами на ближайший год компания JetBrains назвала следующие:

- выпуск стабильного релиза Compose Multiplatform для iOS. В числе целей заявлены не только общие улучшения базового фреймворка, но и интеграции с iOS SDK и повышение производительности;
- улучшенная поддержка мультиплатформенной разработки в IntelliJ-средах не только Kotlin Multiplatform, но и Compose Multiplatform;
- выпуск публичной версии экспорта Kotlin в Swift. Этот механизм позволит конвертировать код из Kotlin в Swift напрямую, без предварительных преобразований в C/Objective-C;
- поддержка кросс-платформенной разработки в системе конфигурации и сборки проектов Amper;
- улучшение и оптимизация разработки кросс-платформенных библиотек для Native-платформ.

Наиболее значимыми и ожидаемыми для большинства разработчиков KMP являются доработки и улучшения, связанные с Compose Multiplatform для iOS и прямым экспортом Kotlin в Swift. Они позволят не только создавать более производительные и оптимизированные приложения для iOS с помощью KMP, но и распространить возможности нативного Android Jetpack Compose на все мобильные таргеты кросс-платформы.

ГЛАВА 1

Создаем и настраиваем проект Kotlin Multiplatform

В этой главе рассмотрим:

- какие инструменты нужны для разработки;
- как их скачать и настроить;
- каково базовое устройство проекта KMP;
- какие существуют альтернативные IDE для работы;
- как превратить нативный проект в кросс-платформенный.

Для того чтобы начать разрабатывать мобильные приложения на Kotlin Multiplatform, нам прежде всего понадобится установить и настроить все инструменты и IDE. Вам потребуется SDK для всех таргетов, которые вы собираетесь поддерживать, то есть Android SDK для Android и iOS SDK, если вы будете писать под iOS.

> **ПРИМЕЧАНИЕ** Учтите, что если вы планируете разрабатывать мобильные приложения под iOS, то вам потребуется компьютер с Mac OS (Macbook Pro или Mac mini), так как на другую систему SDK установить не получится. Для скачивания инструментов вам будет нужен действующий аккаунт Apple Developer (для приобретения переходите на https://developer.apple.com).

Основной и официальной IDE для работы с Kotlin Multiplatform на данный момент является Android Studio от компании JetBrains. Для скачивания свежей версии необходимо перейти на сайт https://developer.android.com и выбрать Android Studio в разделе Develop Get (рис. 1.1).

Затем для скачивания Android Studio последней стабильной версии нажимаем Download Android Studio (рис. 1.2).

> **ВНИМАНИЕ!** Следующий шаг зависит от того, на какую ОС вы будете устанавливать IDE. Для Mac OS необходимо будет выбрать вариант Android Studio, соответствующий архитектуре вашего процессора (рис. 1.3). Обратите внимание, что таргет iOS поддерживается только на Mac OS, и для разработки под него потребуется Xcode.

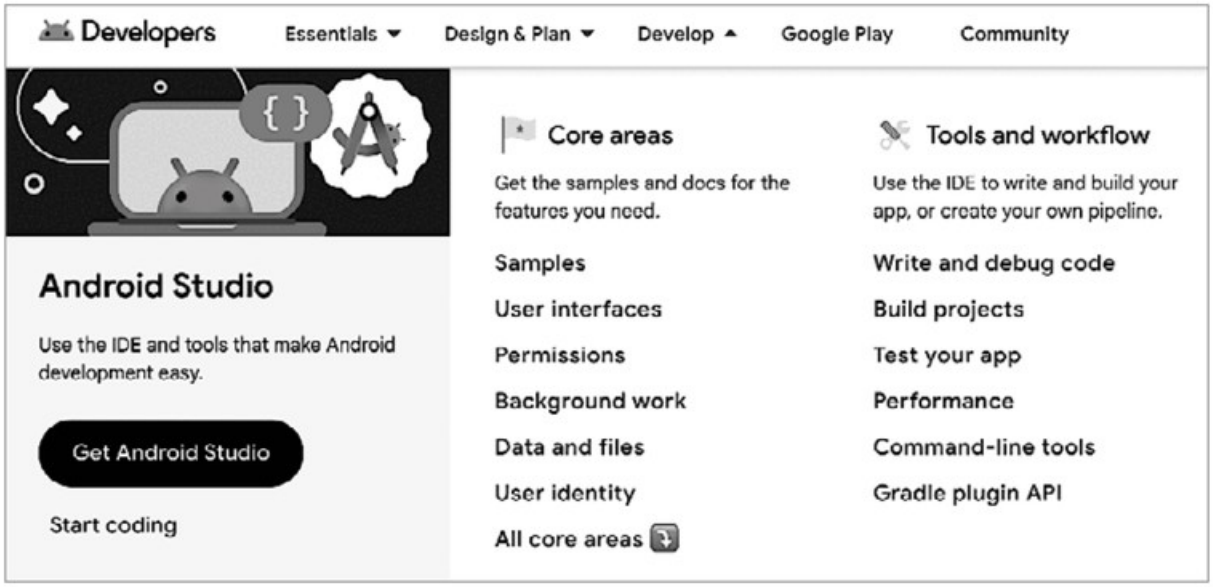


Рис. 1.1. Сайт developer.android.com

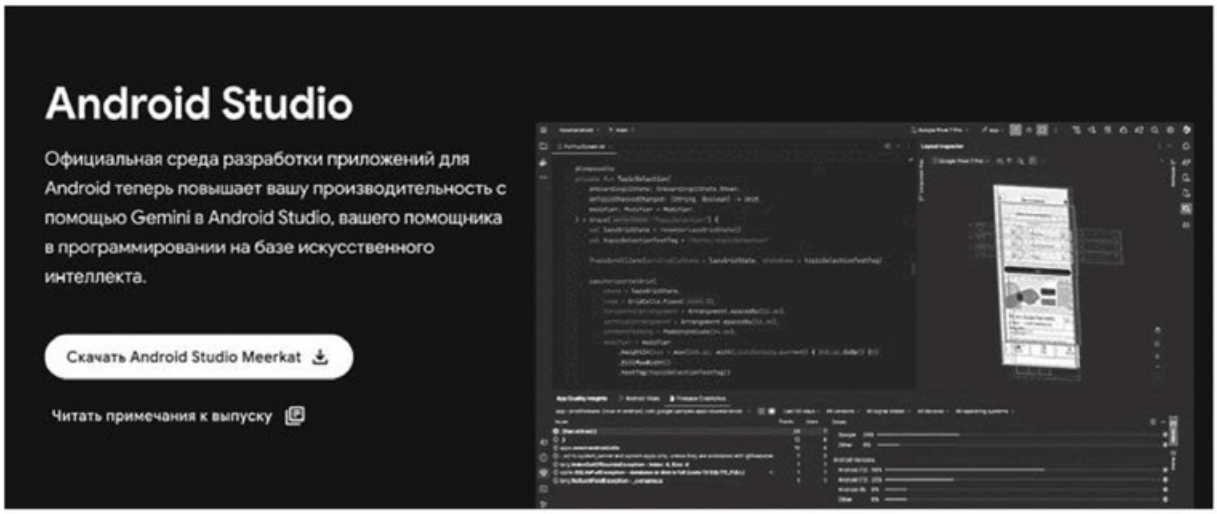


Рис. 1.2. Переходим к скачиванию



Рис. 1.3. Варианты дистрибутива

После того как вы скачали инсталлятор, запускайте его и следуйте инструкциям мастера установки — все стандартно. Если на вашем компьютере еще нет Android SDK, то вам предложат пройти шаг автоматической загрузки и установки компонентов SDK (рис. 1.4).

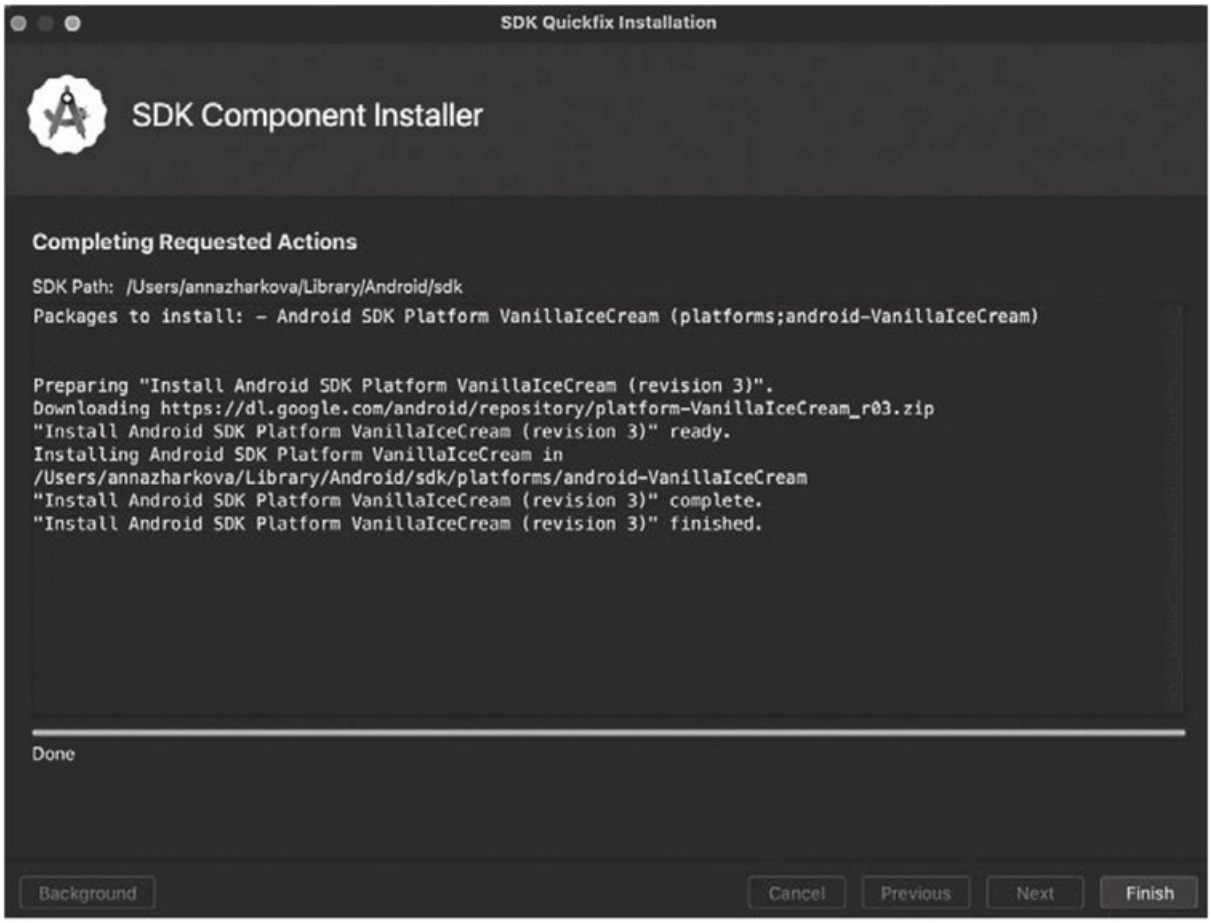


Рис. 1.4. Начало установки Android SDK

После нажатия кнопки Finish вы готовы приступить к работе. Остается одна маленькая деталь — установка плагина Kotlin Multiplatform Mobile. Для этого перейдем в меню Settings ▸ Plugins и на вкладке Marketplace наберем название нужного плагина в строке поиска (рис. 1.5).

Нажмем Apply ▸ OK.

> **ВНИМАНИЕ!** Если по каким-то причинам у вас проблемы с доступом к JetBrains Marketplace и самим библиотекам, попробуйте включить дополнительные настройки прокси сети вашего компьютера.

Если вы собираетесь разрабатывать и под iOS, вам потребуется установить Xcode со встроенным iOS SDK. Для загрузки перейдите на https://developer.apple.com/xcode/ (рис. 1.6).

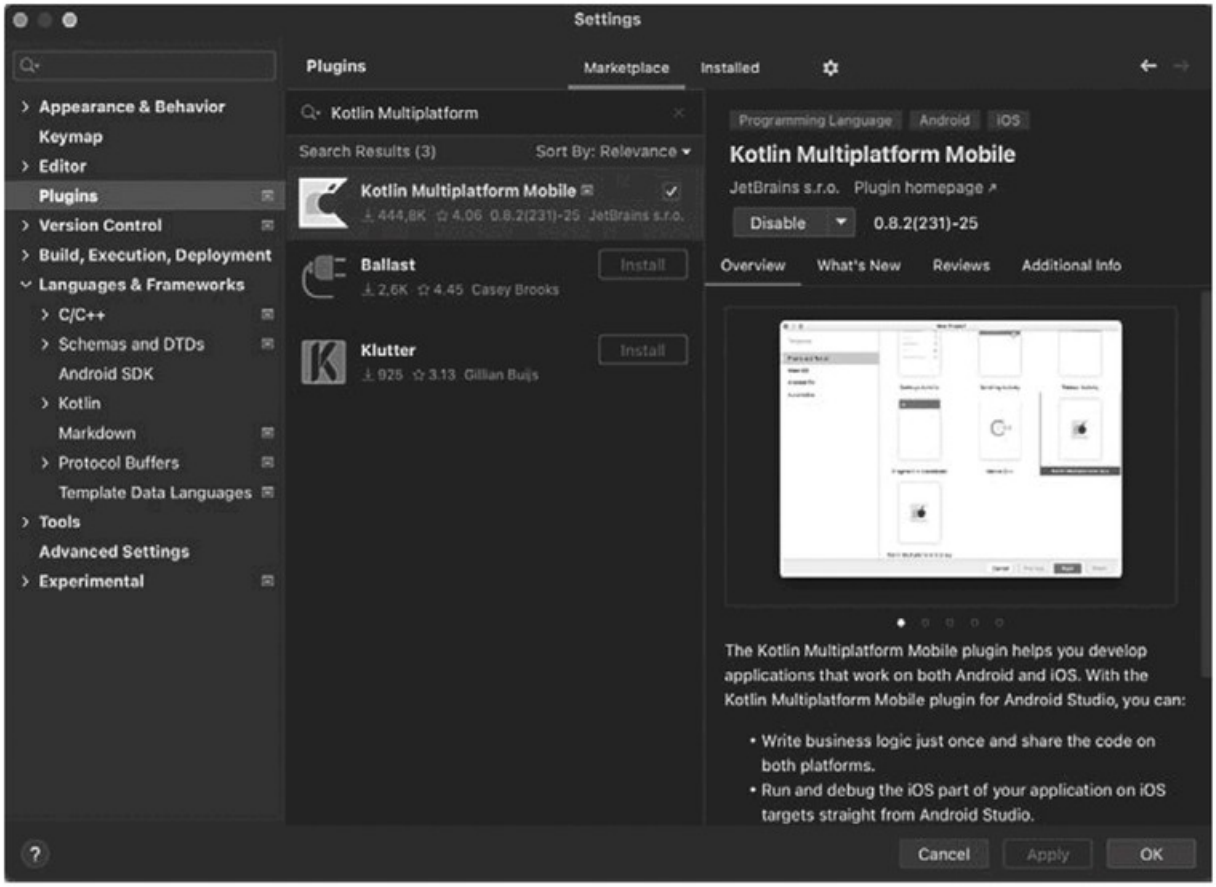


Рис. 1.5. Установка плагина

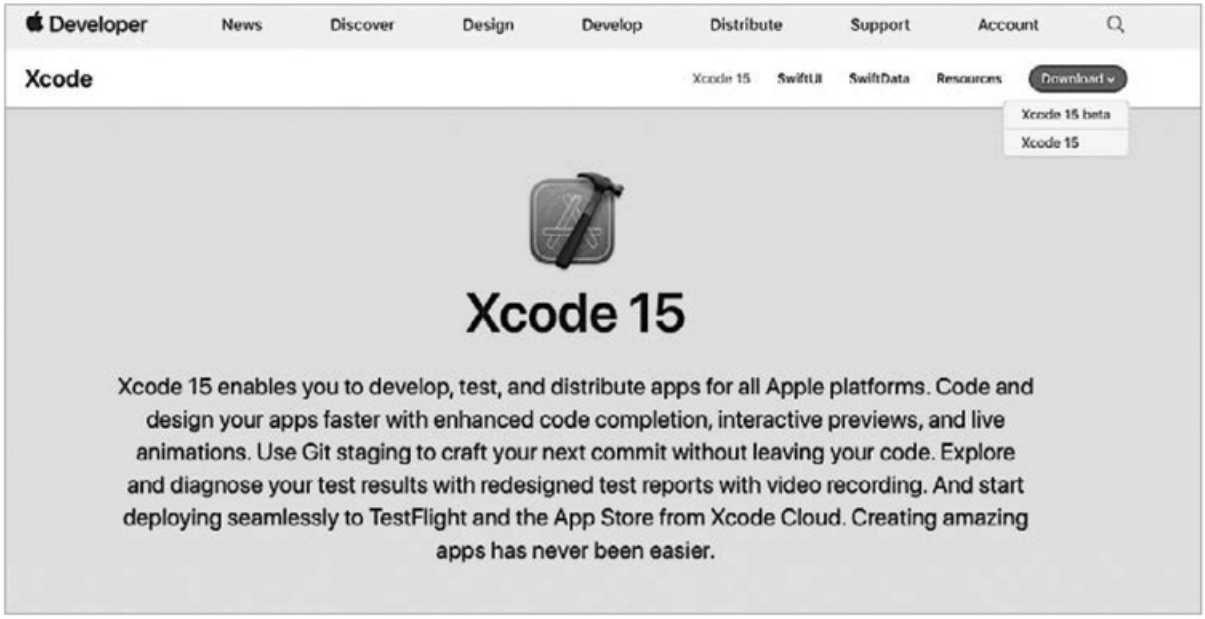


Рис. 1.6. Загружаем Xcode

Обратите внимание, что Kotlin Multiplatform гарантированно работает только со стабильными версиями Xcode. Также стоит иметь в виду, что Xcode можно

установить только на Mac OS. Для компьютера на Linux или Windows потребуется использование подходящей виртуальной машины.

Итак, все скачали, установили, и теперь вы точно готовы к работе.

1.1. Создание проекта с нуля

Начнем с создания приложения с нуля. Для этого переходим в меню File ▸ New ▸ New Project. Плагин KMM (Kotlin Multiplatform Mobile) позволяет нам создать базовое приложение с помощью базового шаблона (рис. 1.7).

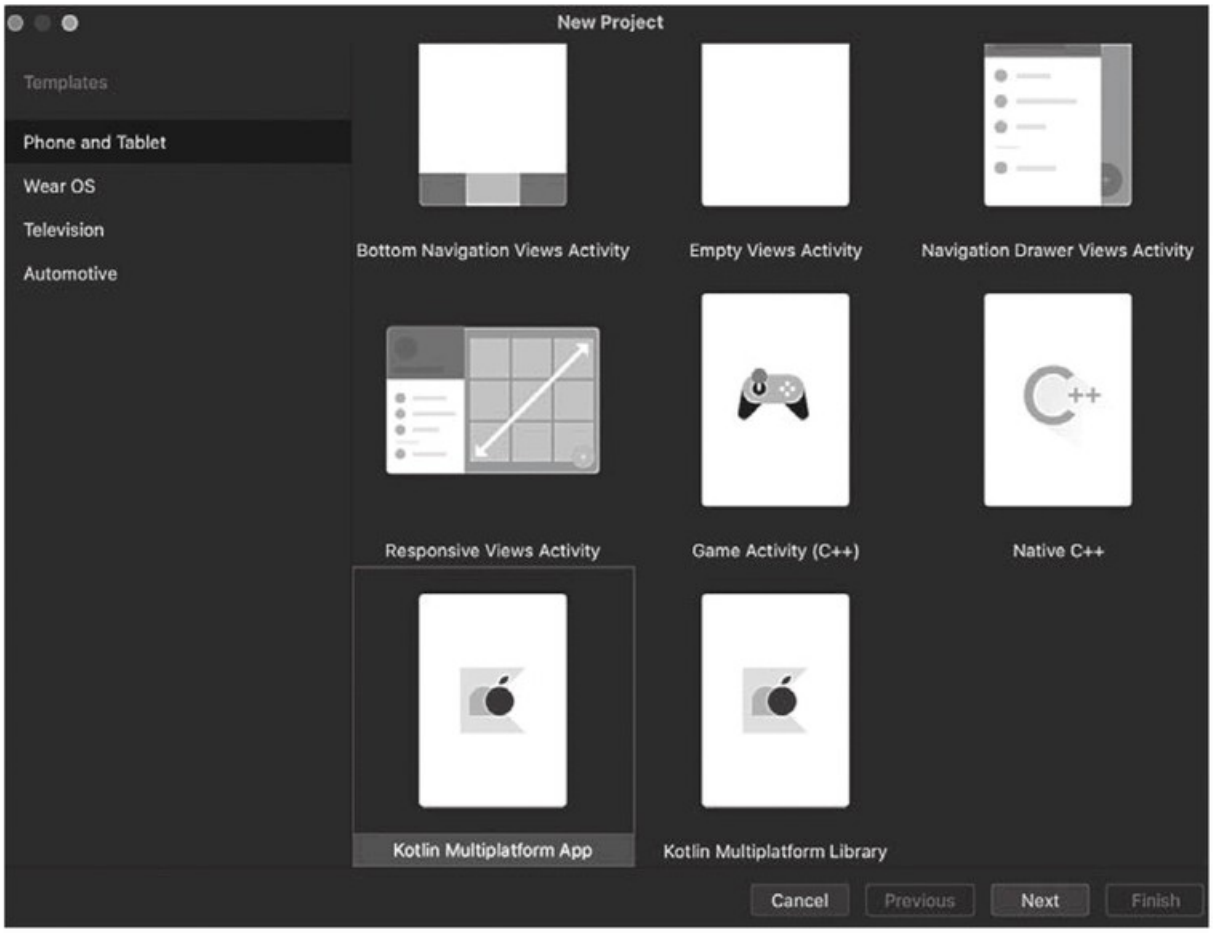


Рис. 1.7. Выбор шаблона приложения

Шаблон Koltin Multiplatform App создаст базовое приложение с несколькими мобильными таргетами. В мастере создания укажем наименование проекта, его расположение, а также целевую версию Android (рис. 1.8).

Далее настроим наименование общего модуля кода, название модулей нативных приложений и выберем способ подключения shared-модуля к iOS-приложению (рис. 1.9).

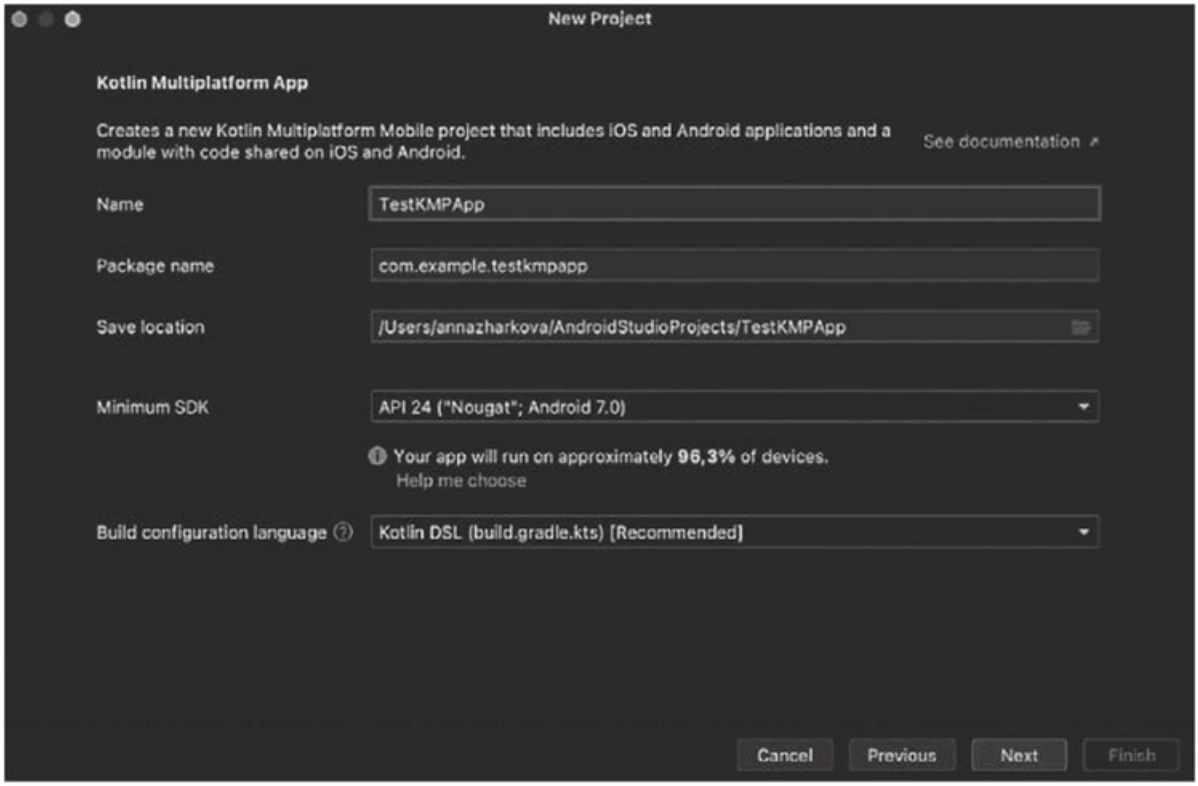


Рис. 1.8. Настройка имени проекта и расположения

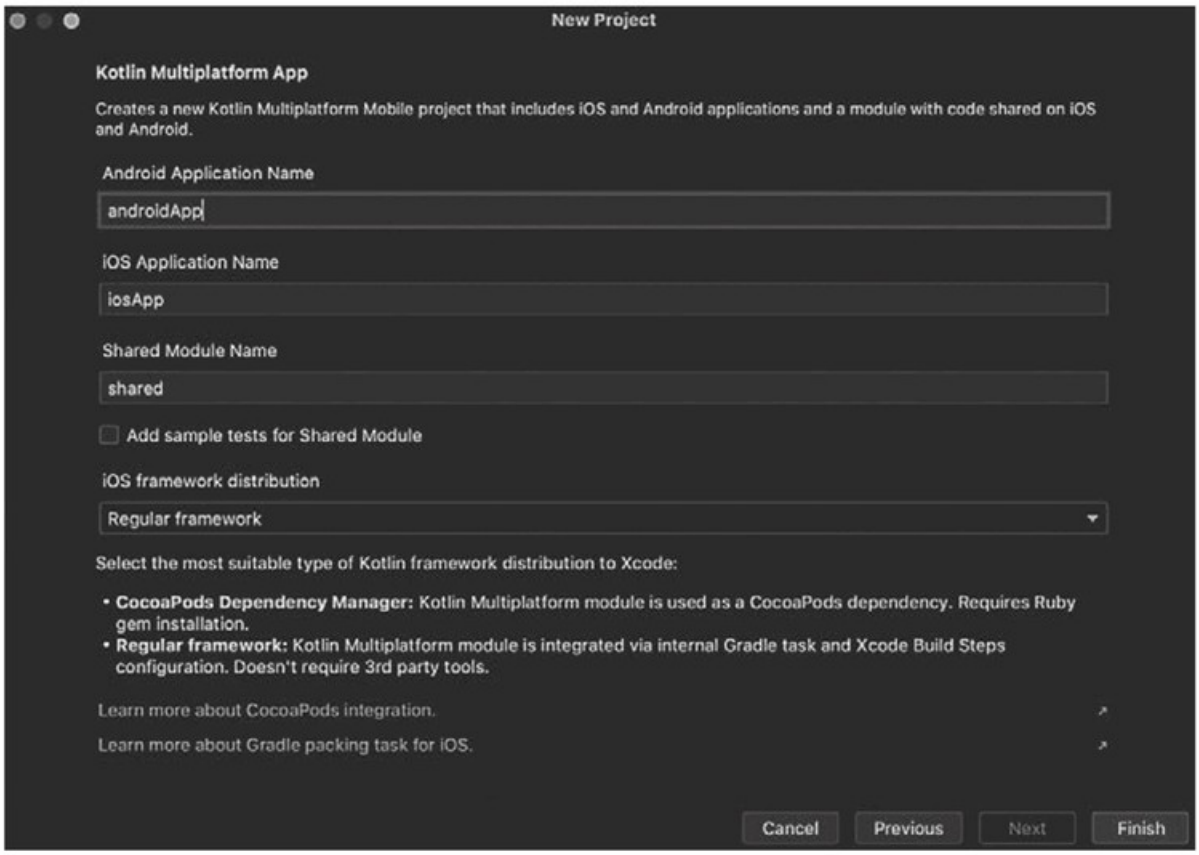


Рис. 1.9. Настройка параметров общего модуля при создании проекта

В настоящее время поддерживается подключение модуля как обычного Xcode-фреймворка или через CocoaPods. Ожидается, что через какое-то время будет возможно подключение и через Swift Package Manager. На момент написания книги соответствующая задача в YouTrack JetBrains еще в статусе Open, но большая часть связанных задач уже закрыта (рис. 1.10).

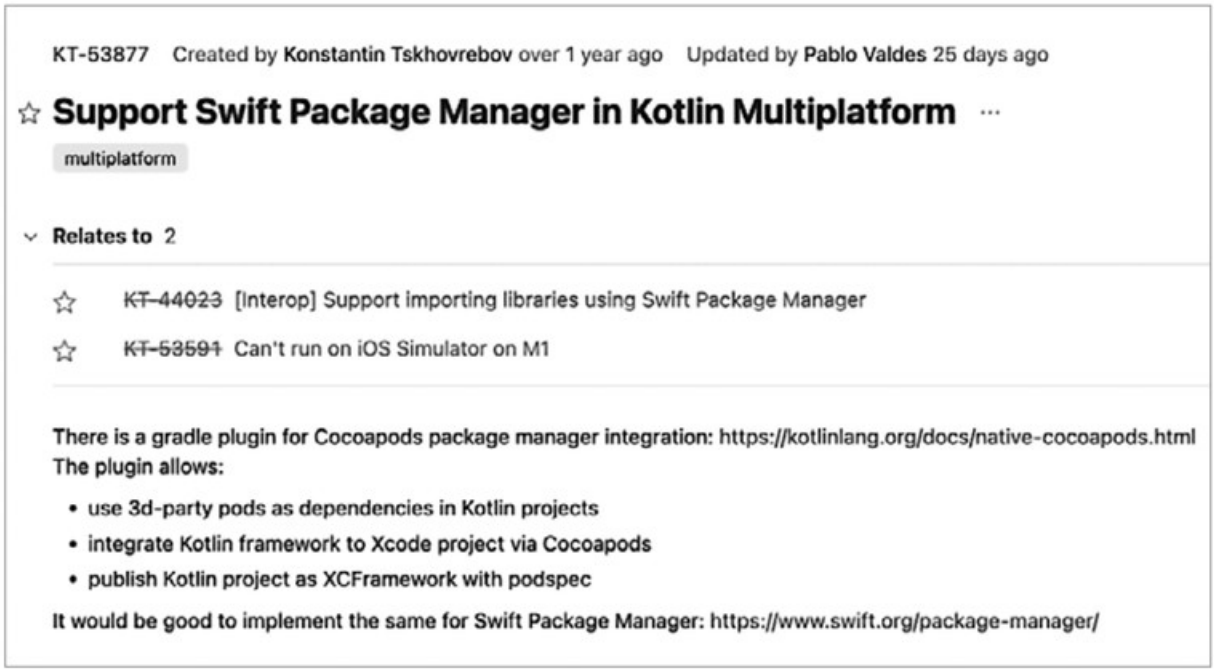


Рис. 1.10. Статус поддержки SPM

Нажмем Finish, и IDE приступит к синхронизации Gradle-зависимостей только что созданного проекта.

Пока происходит первичная настройка проекта, обратим внимание на его структуру. Базовый проект Kotlin Multiplatform состоит из следующих частей (рис. 1.11):

- shared-модуль;
- нативное приложение Android;
- нативное приложение iOS.

iOSApp и androidApp — это обычные нативные приложения iOS и Android, которые используют shared-модуль в качестве библиотеки с бизнес-логикой. Приложение iOS пишется на Swift и может быть реализовано как с помощью UIKit, так и с помощью SwiftUI. Приложение Android также может быть классическим или реализованным с помощью Jetpack Compose. В последних версиях Kotlin Multiplatform базовые нативные приложения реализуются сразу на Jetpack Compose и SwiftUI. Shared-модуль — это и есть, собственно, наш модуль общего кода. Подробнее о его анатомии мы поговорим чуть позже.

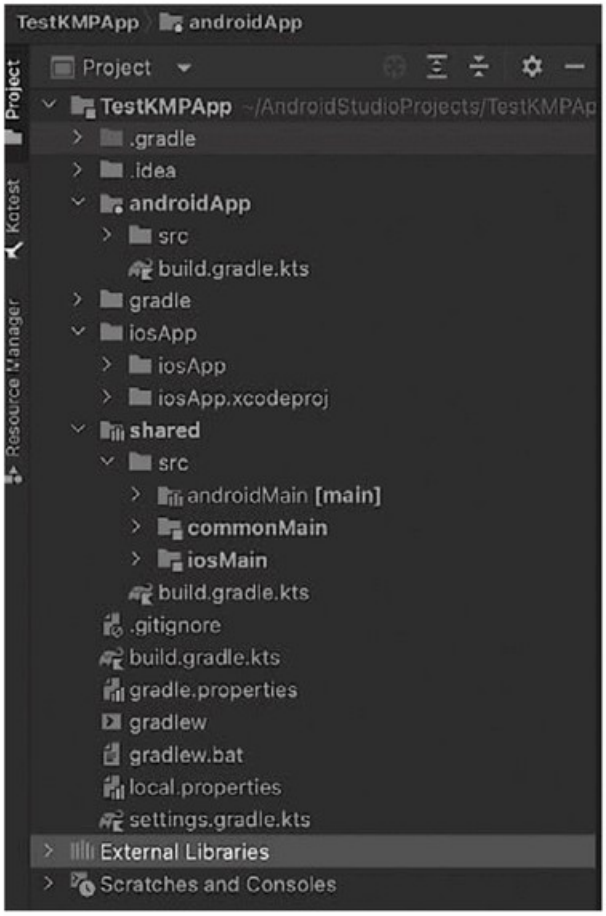


Рис. 1.11. Структура базового приложения KMP

Для корректной работы кросс-платформенного приложения важно его правильно настроить. Поэтому сейчас обратим внимание на особенности настройки и управления зависимостями.

1.2. Настройка проекта и управление зависимостями

В проекте KMP, как и в случае с Android, используется система автоматической сборки Gradle. Необходимые параметры прописываются в соответствующих скриптах `build.gradle.kts`. Такой скрипт есть в каждом модуле приложения. Дополнительно есть скрипт на все приложение для настройки и подключения плагинов (листинг 1.1).

Листинг 1.1. Подключение плагинов в build.gradle.kts

```
plugins {
    //trick: for the same plugin versions in all sub-modules
    alias(libs.plugins.androidApplication).apply(false)
    alias(libs.plugins.androidLibrary).apply(false)
    alias(libs.plugins.kotlinAndroid).apply(false)
    alias(libs.plugins.kotlinMultiplatform).apply(false)
}
```

Рассмотрим внимательнее `build.gradle.kts` shared-модуля. Начинается он с секции подключения плагинов в модуль (листинг 1.2).

Листинг 1.2. Подключение плагинов в shared-модуле

```
plugins {
    alias(libs.plugins.kotlinMultiplatform)
    alias(libs.plugins.androidLibrary)
}
```

По умолчанию подключен плагин Kotlin Multiplatform и плагин для работы Android-таргета как библиотека для Android.

Далее идет секция `kotlin`, в которой описываются все нужные таргеты и подключаются библиотеки для каждого из них (листинг 1.3).

Листинг 1.3. Подключение и настройка таргетов

```
kotlin {
    androidTarget {…}

    listOf(…).forEach {…}

    sourceSets {
        commonMain.dependencies {
            //put your multiplatform dependencies here
        }
        commonTest.dependencies {
            implementation(libs.kotlin.test)
        }
    }
}
```

По умолчанию для проекта указываются таргет Android и таргет(ы) iOS. Для подключения других таргетов (desktop, WASM) потребуется их прописать и настроить вручную.

Для таргета Android по умолчанию указывается поддерживаемая версия JVM (листинг 1.4).

Листинг 1.4. Android-таргет

```
androidTarget {
    compilations.all {
        kotlinOptions {
            jvmTarget = "1.8"
        }
    }
}
```

Таргет Android един для всех устройств и симуляторов. В случае iOS мы рассматриваем таргет под каждую аппаратную архитектуру по отдельности. Вот так выглядит автоматическое подключение библиотеки к iOS-таргетам (листинг 1.5).

Листинг 1.5. Настройка iOS-таргетов

```
listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
).forEach {
    it.binaries.framework {
        baseName = "shared"
        isStatic = true
    }
}
```

Чтобы было удобнее работать с кодом, Kotlin Multiplatform создает для нас единый таргет iOSMain:

```
sourceSets {
        iosMain.dependencies {/*...*/}
    }
```

В блоки `dependencies` мы можем добавлять необходимые зависимости для каждого таргета (листинг 1.6).

Листинг 1.6. Блок sourceSets

```
sourceSets {
    iosMain.dependencies {/*...*/}

    androidMain.dependencies { /*...*/ }

    commonMain.dependencies {
        //put your multiplatform dependencies here
    }
}
```

Мультиплатформенные версии библиотек прописываются в блоке для commonMain. Для установки зависимости строка подключения добавляется в нужный блок, после чего нужно нажать кнопку Gradle sync для синхронизации Gradle:

```
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:*")
```

Вместо * выберите нужную версию библиотеки.

Также можно использовать специальный файл `gradle-lib.versions.iml`, в котором прописать нужные версии библиотек, разбив их на тематические секции (листинг 1.7).

Листинг 1.7. Регистрация версии библиотеки в каталоге Gradle-приложения

```
[versions]
coroutines = "1.10.1"

[libraries]
```

```
coroutines = {
    module = "org.jetbrains.kotlinx:kotlinx-coroutines-core",
    version.ref = "coroutines"
  }
```

Подключение становится более лаконичным (листинг 1.8). Такой подход позволяет вынести управление версиями из `build.gradle.kts`.

Листинг 1.8. Подключение библиотеки из каталога

```
sourceSets {
            commonMain.dependencies {
                implementation(libs.coroutines)
            }
        }
```

В самом низу `build.gradle.kts` — секция для настройки параметров Android-таргета и нативного приложения (листинг 1.9).

Листинг 1.9. Настройки нативного Android-приложения

```
android {
  namespace = "com.example.testkmpapp"
  compileSdk = 34
  defaultConfig {
    minSdk = 24
  }
  compileOptions {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
  }
}
```

Это настройки основного проекта и shared-модуля. Для подключения shared-модуля к нативному Android-приложению необходимо указать путь к нему в зависимостях `build.gradle.kts` основного модуля нативного приложения:

```
dependencies {
    implementation(projects.shared)
}
```

Для iOS-приложения необходимо сделать следующее.

На вкладке Build Settings в Search Path укажем путь для поиска нашего собранного фреймворка:

```
$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)
```

Далее в Linking ▸ General в Other linking flags добавим название модуля (в нашем случае shared) и флаги `$(inherited)` и `-frameworks` (рис. 1.12).



Рис. 1.12. Параметры для нативного iOS-приложения

Чтобы shared-модуль автоматически собирался в `xcodeframework`, необходимо прописать скрипт. На вкладке Build Phases (рис. 1.13) добавим Run script с кодом:

```
cd "$SRCROOT/.."
./gradlew :shared:embedAndSignAppleFrameworkForXcode
```

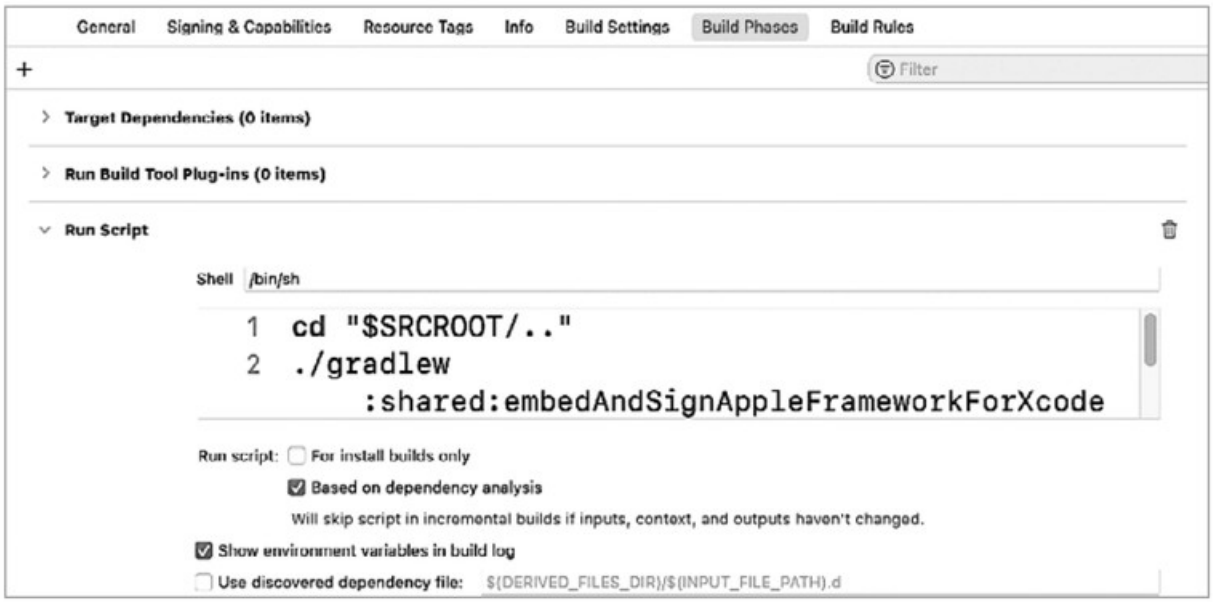


Рис. 1.13. Подключение скрипта в Build Phases

С помощью указанного скрипта мы обращаемся к задаче Gradle для создания и встраивания библиотеки в нативное iOS-приложение.

После запуска сборки на стороне iOS-приложения или конфигурации iOSApp из проекта Kotlin Multiplatform в папке build shared-модуля появится скомпилированный `xcodeframework` (рис. 1.14).

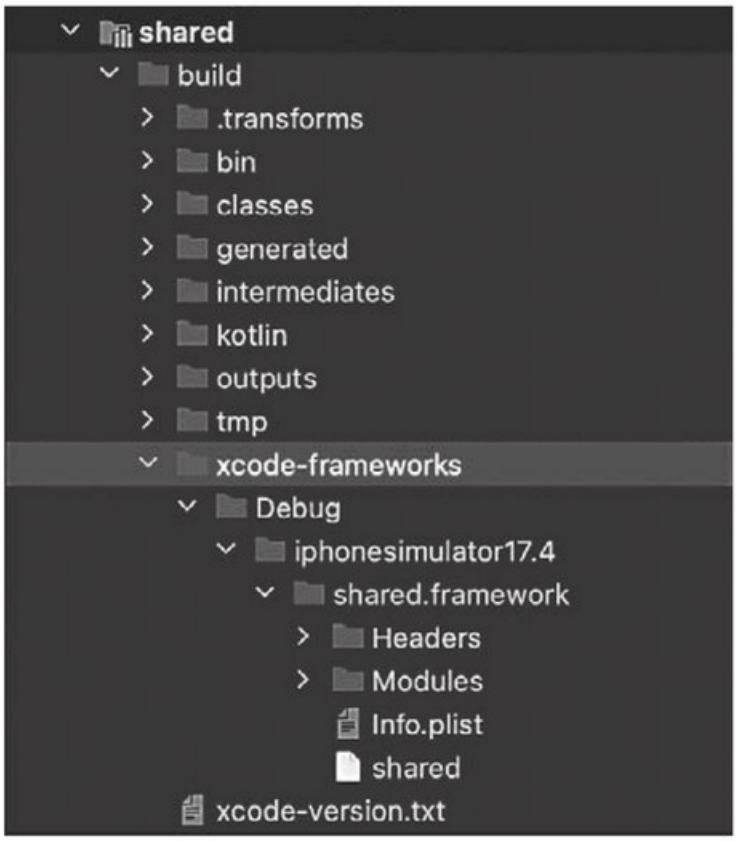


Рис. 1.14. Структура xcodeframework

Прописывать пути к библиотеке и указывать дополнительные настройки вручную потребуется в случае, если мы подключаем shared-модуль к существующим нативным приложениям Android и iOS. При создании проекта KMP через визард это будет сделано автоматически.

Итак, проект настроен. Теперь мы можем больше внимания уделить нашему shared-модулю.

1.3. Shared-модуль приложения

Основное назначение shared-модуля — предоставлять код, общий для разных таргетов. Разные таргеты под капотом связаны с разными нативными библиотеками платформ. Поэтому сам модуль не цельный, а состоит из commonMain-части, где находятся основной общий код и код платформенных частей (в случае мобильных таргетов iOSMain и androidMain), в которых можно реализовать что-нибудь с учетом особенностей платформы, либо код только для какого-то отдельного таргета (рис. 1.15).

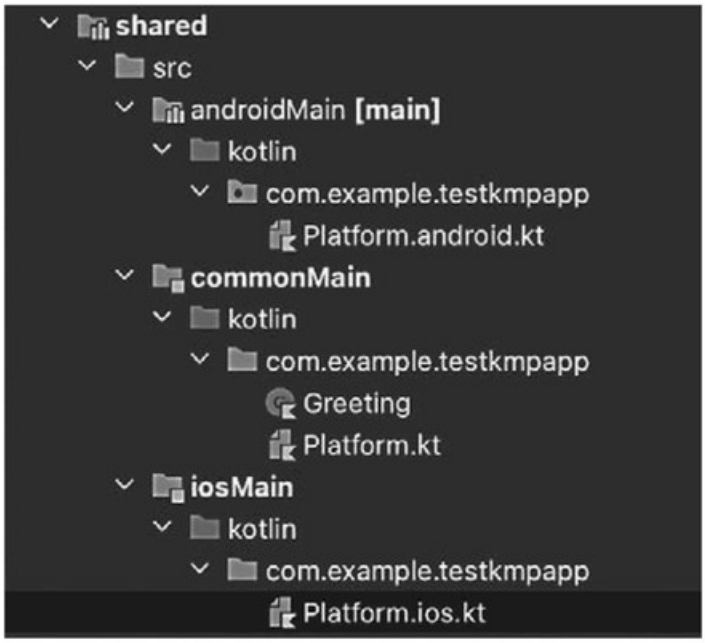


Рис. 1.15. Структура shared-модуля

Как это работает? Дело в том, что в каждой из частей shared-модуля используется своя версия Kotlin, подходящая под указанный таргет: общая версия для commonMain, Kotlin JVM — для Java и Android (androidMain), Kotlin Native — для iOS (iOSMain), Linux, Mac OS, Kotlin JS/Wasm — для веб (рис. 1.16).

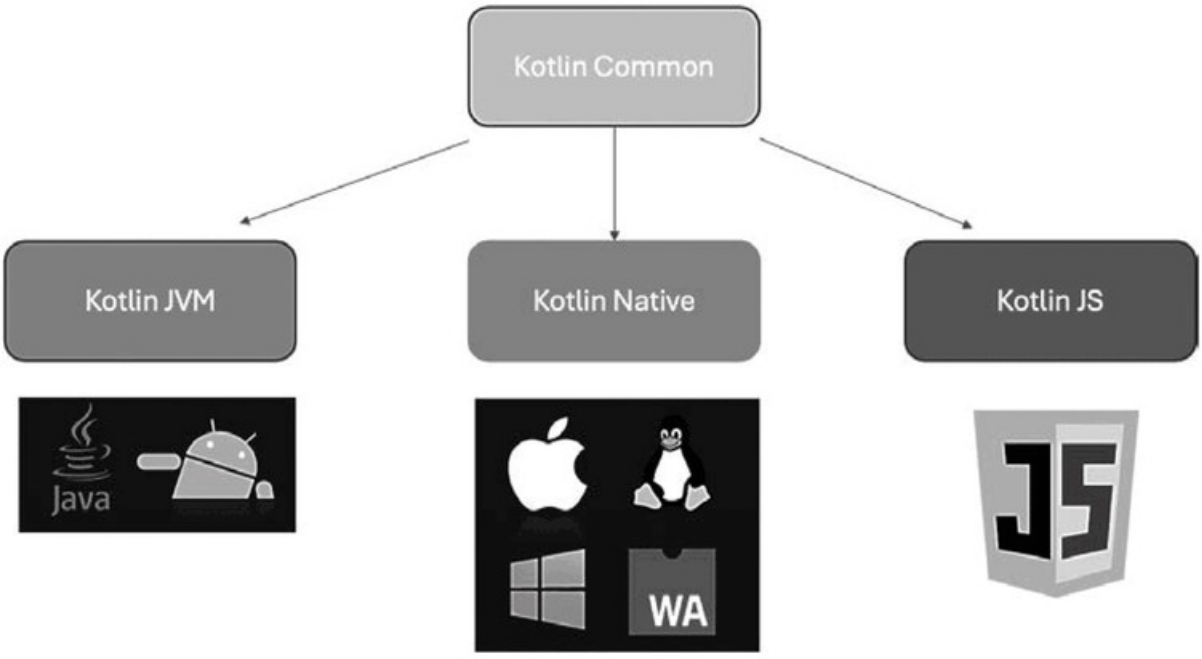


Рис. 1.16. Версии Kotlin для разных таргетов

Единой входной точкой для обращения к коду shared-модуля из других модулей или нативных приложений являются публичные методы и свойства

из commonMain. Для того чтобы бесшовно подставлять нужные реализации из iOSMain/androidMain, существует механизм `expect/actual`. Декларации в commonMain помечаются как `expect`, то есть ожидаемые для реализации, а собственно имплементации кода в платформенных таргетах — как `actual` (рис. 1.17).

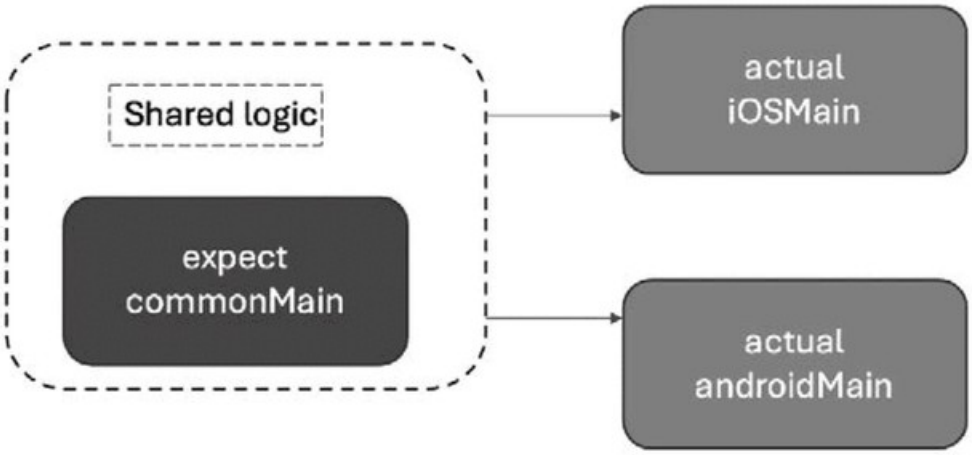


Рис. 1.17. Соотношение expect/actual

Давайте рассмотрим пример кода базового приложения (рис. 1.18).

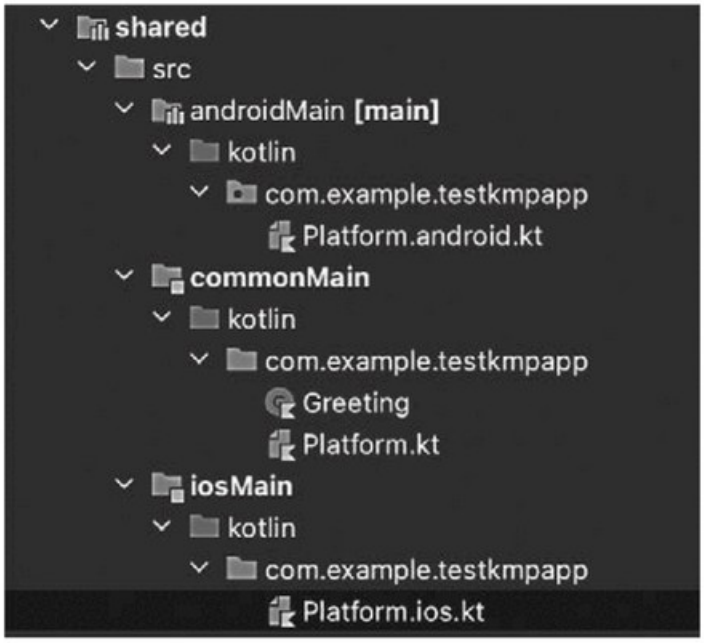


Рис. 1.18. Базовое приложение Kotlin Multiplatform

В commonMain в `Platform.kt` у нас описаны интерфейс `Platform` и функция `getPlatform`, помеченная `expect` (листинг 1.10).

Листинг 1.10. Декларация кода в commonMain

```
package com.example.testkmpapp

interface Platform {
  val name: String
}

expect fun getPlatform(): Platform
```

Каждый из таргетов обязательно должен содержать `actual`-реализацию данного метода (листинг 1.11).

Листинг 1.11. Actual-реализация в iOS

```
package com.example.testkmpapp

import platform.UIKit.UIDevice

class IOSPlatform: Platform {
  override val name: String = UIDevice.currentDevice.systemName() + " " +
UIDevice.currentDevice.systemVersion
}

actual fun getPlatform(): Platform = IOSPlatform()
```

Обратите внимание, что в iOSMain нам доступны нативные библиотеки iOS SDK (Foundation, UIKit, Network и т. п.). В нашем примере через `UIKit.UIDevice` мы получаем название ОС и ее версию. Подробнее о механизме связи iOS SDK и shared-модуля и взаимодействия Swift с Kotlin читайте в следующей главе.

Код для Android также содержит специфические для платформы конструкции. В нашем примере мы получаем данные об ОС через `Build.VERSION` (листинг 1.12).

Листинг 1.12. Actual-реализация в Android

```
//android

package com.example.testkmpapp

class AndroidPlatform : Platform {
  override val name: String = "Android ${android.os.Build.VERSION.SDK_INT}"
}

actual fun getPlatform(): Platform = AndroidPlatform()
```

Посмотрим, как общий код вызывается из нативных приложений. На стороне iOS shared-модуль подключается в приложение на Swift как обычный фреймворк. В этом случае код вызывается из `ContentView` (листинг 1.13).

Листинг 1.13. Вызов со стороны нативного iOS-приложения

```
import SwiftUI
import shared

struct ContentView: View {
       let greet = Greeting().greet()

       var body: some View {
         Text(greet)
       }
}
```

Проект Kotlin Multiplatform содержит базовое нативное iOS-приложение, написанное на актуальном стеке SwiftUI. Подробно подготовку iOS-приложения разберем в следующей главе.

В данном случае у нас просто текстовый контрол `Text`, внутрь которого передается значение строки, полученное из общего модуля, а именно функции `greet` класса `Greeting`.

На стороне Android наше приложение написано на Jetpack Compose. Визуальные компоненты реализованы с помощью специальных функций `Composable`, которые подключаются к основному классу экрана приложения MainActivity. Внутрь `Composable GreetingView` передается текст, также полученный из `Greeting().greet()`, как в листинге 1.14.

Листинг 1.14. Вызов со стороны Android-приложения

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MyApplicationTheme {
        Surface(
          modifier = Modifier.fillMaxSize(),
          color = MaterialTheme.colorScheme.background
        ) {
                    // Composable с текстовым контролом
          GreetingView(Greeting().greet())
        }
      }
    }
  }
}
```

Теперь давайте проверим, как это работает, и запустим наш код. Если у вас установлен Xcode нужной версии, то Android Studio позволит вам запустить

конфигурации для Android- или iOS-приложений. Эмулятор для запуска или устройство для Android-конфигурации (по умолчанию androidApp) мы можем указать прямо в панели (рис. 1.19).

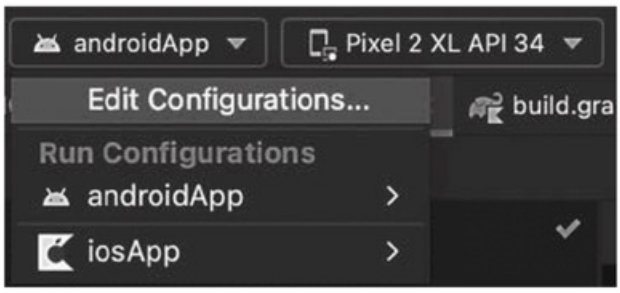


Рис. 1.19. Выбор конфигурации и запуск приложения

Для того чтобы выбрать устройство для запуска или отладки iOSApp, необходимо открыть Edit Configurations и указать нужный таргет (рис. 1.20).

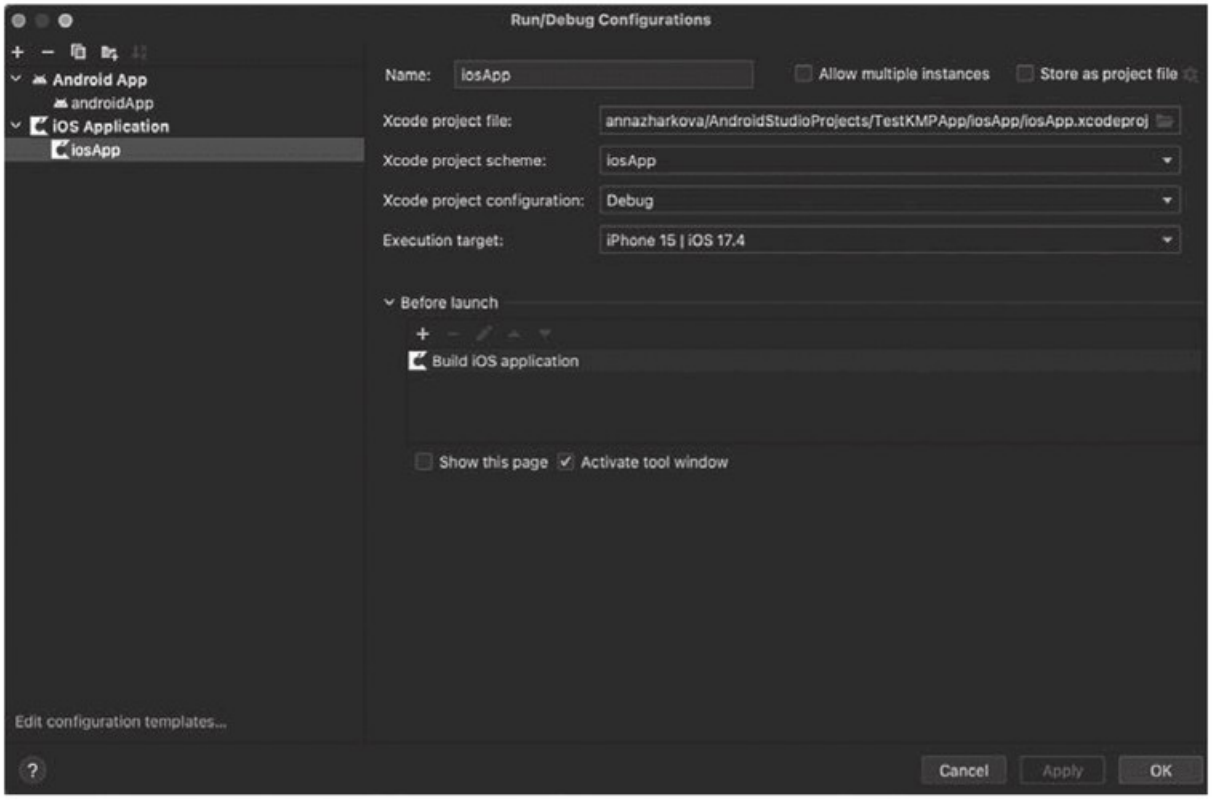


Рис. 1.20. Настройки конфигурации для запуска iOS

Также можно запустить последовательно обе конфигурации (рис. 1.21).



Рис. 1.21. Запуск таргетов одновременно

Как мы видим, на каждой из платформ у нас вывелась строка с данными, корректными для выбранного таргета.

Если вы никогда не имели дела с разработкой под Android (приветствуем смелых айосеров!), вам может потребоваться создать эмулятор для отладки и запуска приложения. Для этого зайдите в меню Tools ▸ Device SDK Manager и выберите нужный виртуальный девайс или создайте собственный, следуя инструкциям мастера (рис. 1.22).

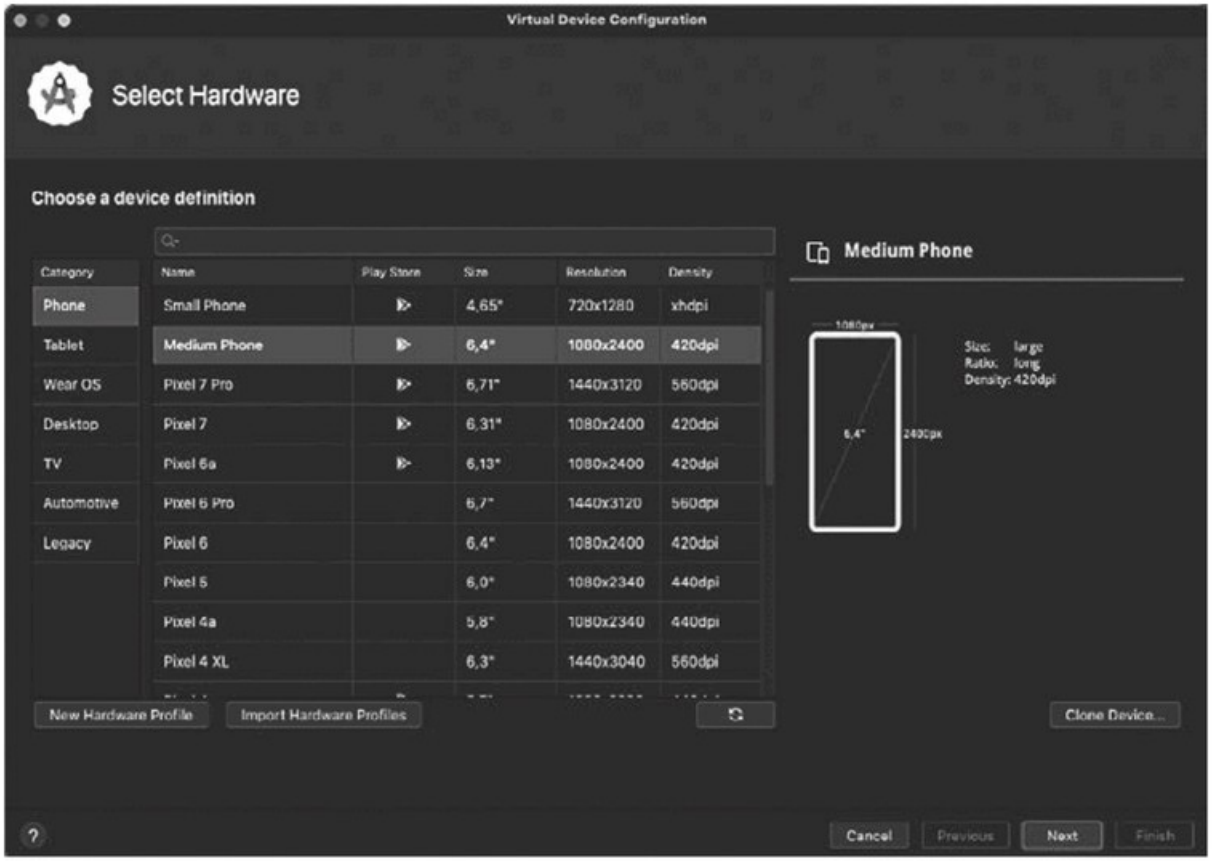


Рис. 1.22. Создание эмулятора

1.4. Создание проекта из существующего нативного

Часто кросс-платформенное приложение должно базироваться на существующем нативном Android- или iOS-приложении. Нам останется только добавить shared-модуль, все подключить и настроить. Есть несколько способов решить эту задачу.

1. Создать обычный проект Kotlin Multiplatform, заменить базовые androidApp и iOSApp на наши нативные проекты, в `settings.gradle` указать название и путь к модулю Android-приложения (листинг 1.15).

Листинг 1.15. Подключение модулей

```
rootProject.name = "TestKMPApp"
include(":androidApp")
include(":shared")
```

Затем подключить shared-модуль к Android и iOS, как мы делали ранее. На Android нужно добавить ссылку на shared-модуль в блок зависимостей:

```
dependencies {
    implementation(projects.shared)
}
```

Для iOS добавить Run script в Build Phases, прописать путь к shared-модулю и build-каталогу, куда сгенерируется `xcodeframework`, а также флаги компиляции и линковки (см. рис. 1.12 и 1.13).

2. Создать библиотеку Kotlin Multiplatform и подключить ее к нативным проектам. Для этого потребуется выбрать соответствующий шаблон проекта при его создании (рис. 1.23).

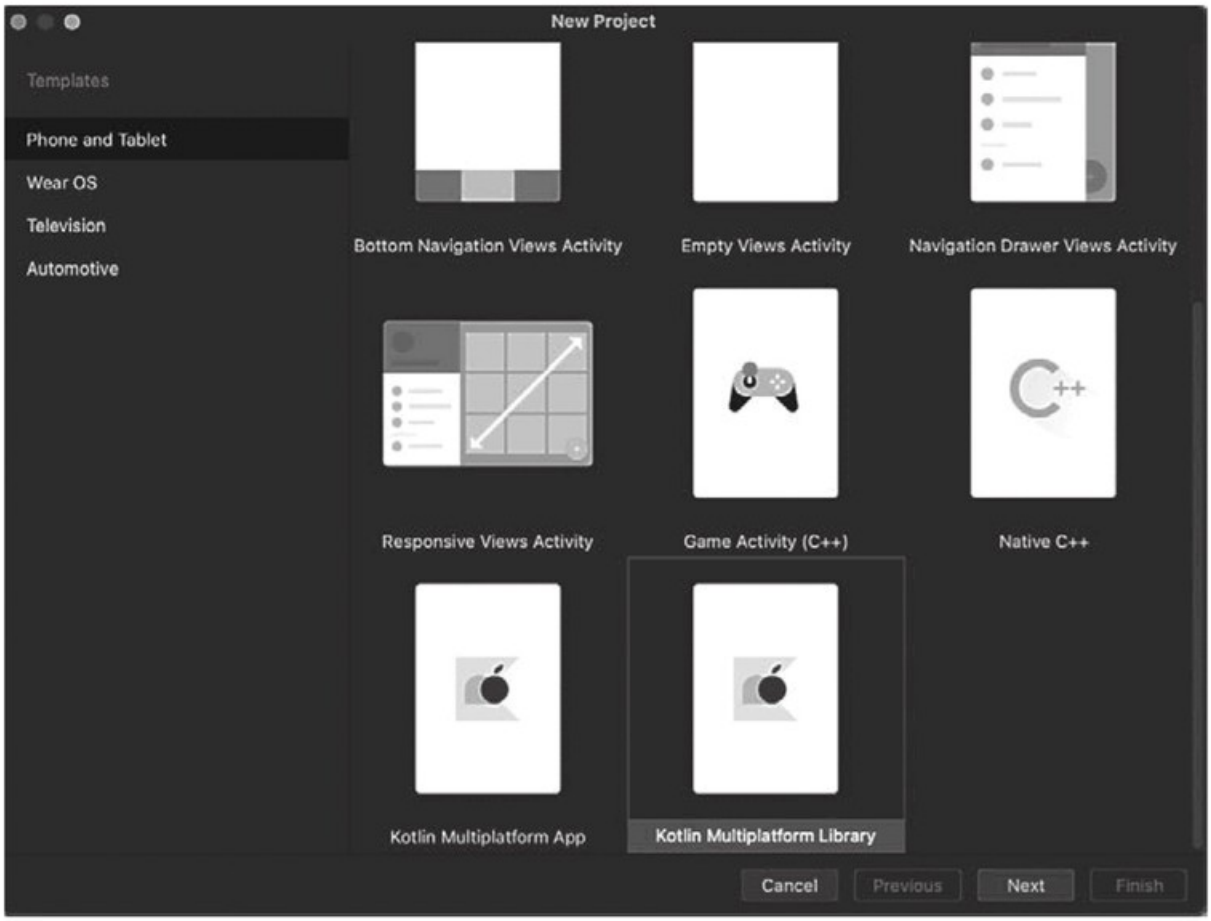


Рис. 1.23. Создание библиотеки

Полученный проект будет содержать только shared-модуль со всей структурой и таргетами. Нативные приложения также потребуется подключать вручную, как описано выше в пункте 1.

Полный пример создания и использования KMP-библиотеки в кросс-платформенном приложении см. в главе 2.

3. Создать новый модуль в Android-приложении. Мы можем пойти путем самурая и добавить новый модуль в проект как Kotlin Library. Все настройки придется указать вручную, как в пункте 1.

Лучше и удобнее будет скопировать их из существующего приложения Kotlin Multiplatform.

Это шаги, нужные для создания структуры приложения Kotlin Multiplatform из нативного приложения. В следующих главах мы начнем рассматривать особенности взаимодействия iOS и Kotlin-кода, а также особенности и специфику разработки слоев логики для кросс-платформенного проекта. Инструкции по главам можно использовать как при разработке нового, так и при миграции существующего нативного приложения. Чек-лист (он же Гайд-шпаргалка) по миграции нативного приложения (Android и iOS) см. в главе 14.

1.5. Пробуем Fleet

Помимо Android Studio JetBrains выпускает еще одну IDE, которую можно использовать для разработки под Kotlin Multiplatform. Это легковесная среда разработки Fleet.

В феврале 2025 года компания JetBrains объявила о прекращении поддержки этой IDE для работы с KMP в течение трех месяцев, то есть до лета 2025 года. Официально рекомендуемые инструменты для работы с KMP — это Android Studio и Intelij Idea. Однако ради интереса вы можете попробовать старые версии Fleet.

Установить ее можно через специальную утилиту JetBrains Toolbox (рис. 1.24).

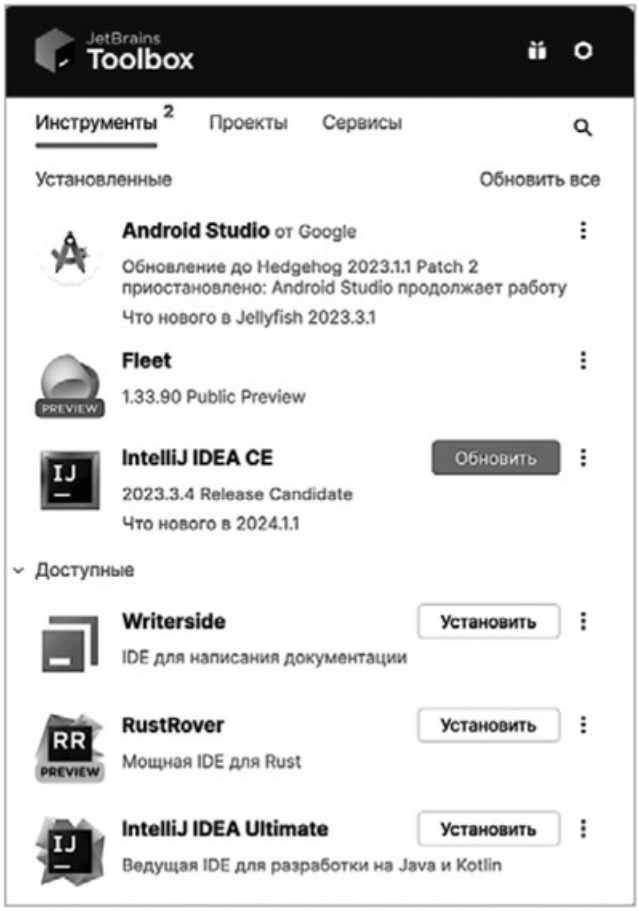


Рис. 1.24. JetBrains Toolbox

На момент написания книги IDE Fleet не является автономной средой, поэтому для полноценной работы вам потребуется установить:

- Android Studio IDE, Android SDK;
- Xcode IDE.

Полную информацию о настройке вы найдете на сайте JetBrains (https://www.jetbrains.com/help/kotlin-multiplatform-dev/fleet.html).

1.5.1. Kotlin Multiplatform Wizard

У Fleet нет встроенного визарда для создания проектов. Вы можете добавлять новые файлы в готовый проект с уже настроенной структурой и редактировать код, но для создания проекта потребуется воспользоваться онлайн-мастером, как на рис. 1.25 (https://kmp.jetbrains.com/).

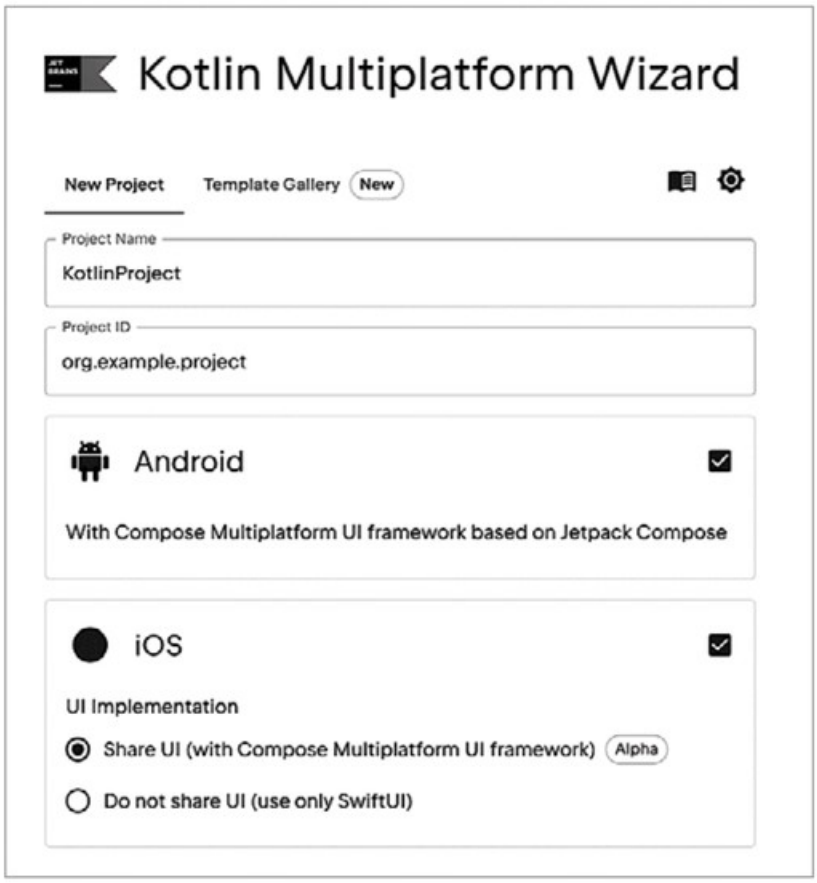


Рис. 1.25. Kotlin Multiplatform Wizard

С помощью этого инструмента вы можете настроить стартовый базовый проект, выбрав желаемые таргеты, и включить поддержку общего UI (подробнее о Compose Multiplatform см. главу 10 этой книги). Также вы можете выбрать готовый шаблон проекта из галереи и скачать его.

Разумеется, Kotlin Multiplatform Wizard подойдет для упрощенной генерации проекта под Android Studio. Возможно, вам этот вариант покажется более удобным.

Можно настроить работу и с удаленным гит-репозиторием.

Откроем скачанный проект и приступим к работе (рис. 1.26).

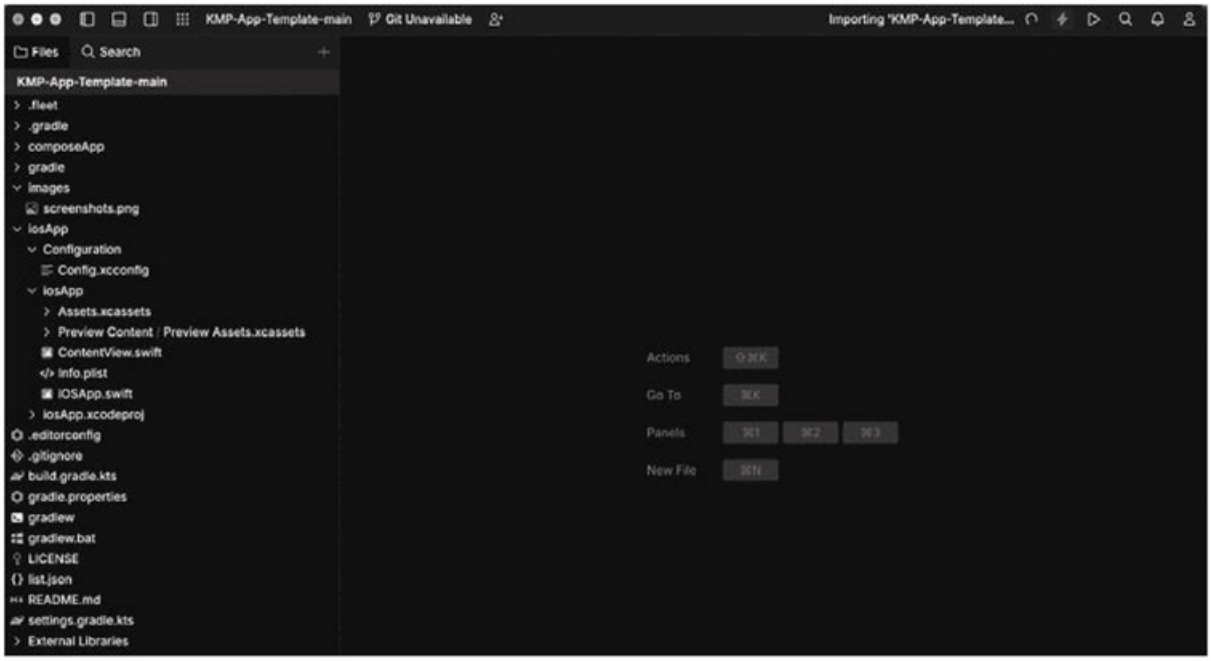


Рис. 1.26. Окно работы в Fleet

Импорт проекта и его индексация в Fleet могут занимать больше времени, чем в Android Studio IDE. Синхронизация зависимостей Gradle включается автоматически при добавлении нового контента.

Для запуска проекта нажимаем кнопку Run либо заходим через соответствующее меню (рис. 1.27).



Рис. 1.27. Запуск приложения

Далее мы можем выбрать конфигурацию для запуска и эмулятор.

ПОДВЕДЕМ ИТОГИ

- ✓ Основным инструментом разработки на данный момент является Android Studio.
- ✓ Также возможна работа с альтернативной IDE Intelij Idea.
- ✓ Если вы планируете разрабатывать проект еще и под iOS, то вам потребуется компьютер на Mac OS, Xcode и аккаунт Apple Developer.
- ✓ Другие решения и IDE для написания приложений не взлетели. Все усилия разработчики технологии направляют на различные улучшения работы в IDE на базе Intelij Idea.

В следующей главе мы составим план, по которому начнем разрабатывать наше учебное приложение и изучать материал, а также создадим для него заготовку. Кроме этого мы подберем архитектуру будущего проекта.

ГЛАВА 2

Подготовка приложения на KMP. Старт разработки

В этой главе рассмотрим:

- план действий и горизонт задач;
- каркас бизнес-логики;
- нативный UI для приложения KMP;
- вариации архитектуры.

Чтобы освоение KMP шло максимально успешно, мы пойдем с вами по маршруту и будем совмещать теорию с закреплением на практике. В качестве максимально емкого и показательного примера выберем приложение для чтения новостей.

Наше приложение будет состоять из двух экранов:

- списка новостей;
- экрана деталей новости.

Каждый из экранов мы реализуем с помощью общего паттерна MVVM сначала с использованием нативных UI обеих платформ, а затем с использованием общего UI на Compose Multiplatform. Также посмотрим на альтернативные решения для данной задачи.

`ViewModel` мы сначала реализуем собственными силами, а потом посмотрим на возможности общей `ViewModel` для KMP от Google из набора решений Jetpack.

Данные мы будем получать с помощью API newsapi.org. Это открытый API для бесплатного использования в учебных и пет-проектах. Для решения этой задачи мы напишем сетевой слой, где подключим сетевой клиент из библиотеки Ktor.

Также мы добавим сохранение понравившейся или заинтересовавшей новости в локальное избранное. Для этого будем использовать библиотеку Room для KMP опять же из Jetpack от Google.

Для работы с бизнес-логикой и совершения длительных операций нам потребуется настроить многопоточную обработку и подключить библиотеку `Coroutines`.

Для построения общей структуры приложения и настройки связи между слоями мы реализуем Dependency Injection.

В общем модуле KMP приложения есть общая логика, а есть платформо-специфичная. Разберем, как правильно работать с такой логикой и как в соответствии с ней писать. Важно правильно распределить усилия по наполнению каркаса нашего приложения и связей между слоями, а также уметь выбрать правильный подход к разработке и использованию компонентов кросс-платформенного приложения.

План довольно плотный и насыщенный! Начнем с самого простого и того, что не требует сложной логики и объяснений. Займемся архитектурой нашего приложения и его каркасом.

2.1. Каркас приложения

В главе 1 мы разобрали, как создать приложение Kotlin Multiplatform, и даже запустили базовый пример. Теперь нам нужно создать учебный проект и подготовить его к дальнейшей работе. И начнем мы с архитектуры, так как это, по сути, и есть каркас любого приложения. Мы не будем разбирать принципы SOLID и другие правила разработки программного обеспечения, также мы не будем обосновывать необходимость создания архитектуры приложения как таковой. Это база. Перейдем к практике.

Если отойти от того факта, что мы делаем кросс-платформенное приложение, то для нативного мы бы его разбили на следующие слои: бизнес-логика для работы с сетью и хранилищем и модули экранов со своей архитектурой (MVx или UDF).

На начальном этапе наше приложение состоит из модуля общего кода и нативных приложений Android и iOS (рис. 2.1).

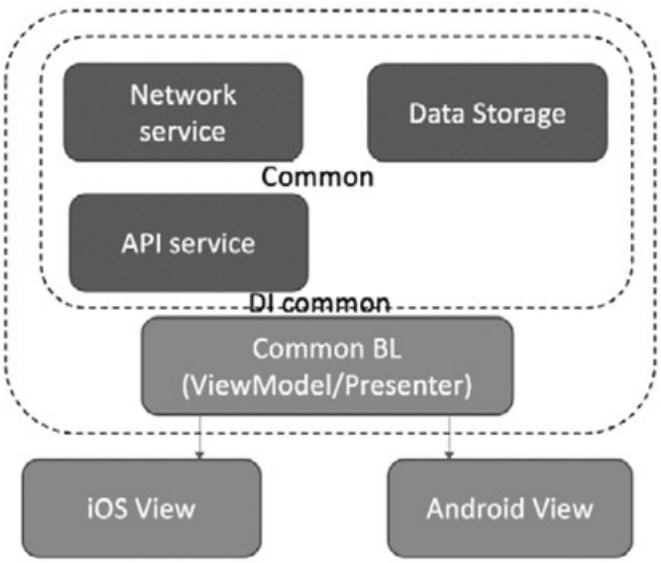


Рис. 2.1. Целевая базовая архитектура

Это та архитектура, которую мы реализуем сейчас. В дальнейшем мы перейдем и на общий UI.

Согласно концепции KMP, в общем модуле размещается то, что не зависит от нативной части и/или возможно для реализации с помощью Kotlin Common, Kotlin Native и Kotlin JVM. В нашем случае это:

- модели данных;
- сетевой клиент;
- хранилище данных.

Начнем с моделей данных. Будем работать, как и планировали, с newsapi.org (рис. 2.2).

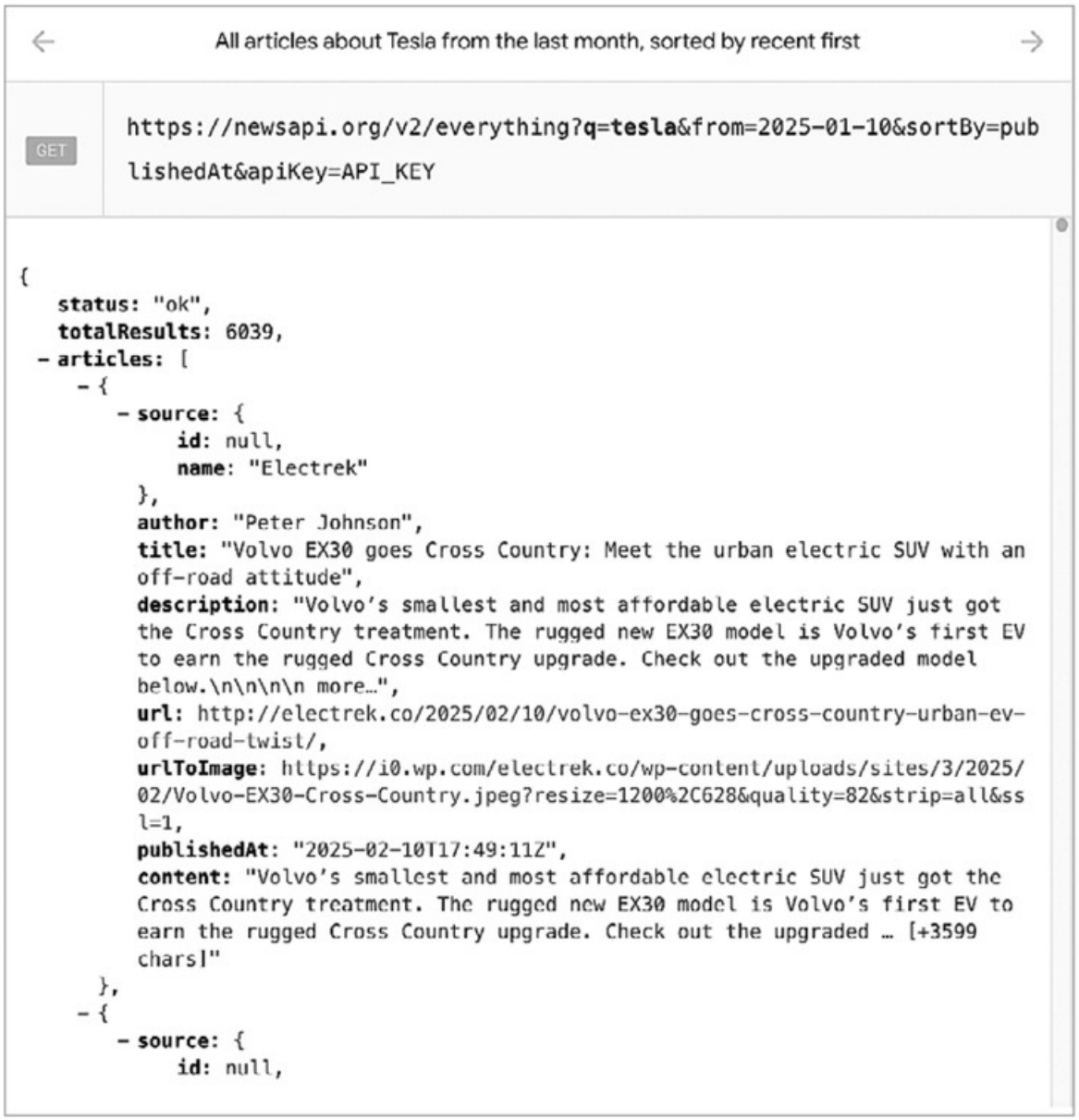


Рис. 2.2. Пример ответа API news.api.org

Для работы с данными нам понадобятся два класса Kotlin: для списка новостей (`articles`) и для единичной новости. При желании можно добавить также структуру данных и для источника (`source`). Разместим их в каталоге data, как на рис. 2.3.

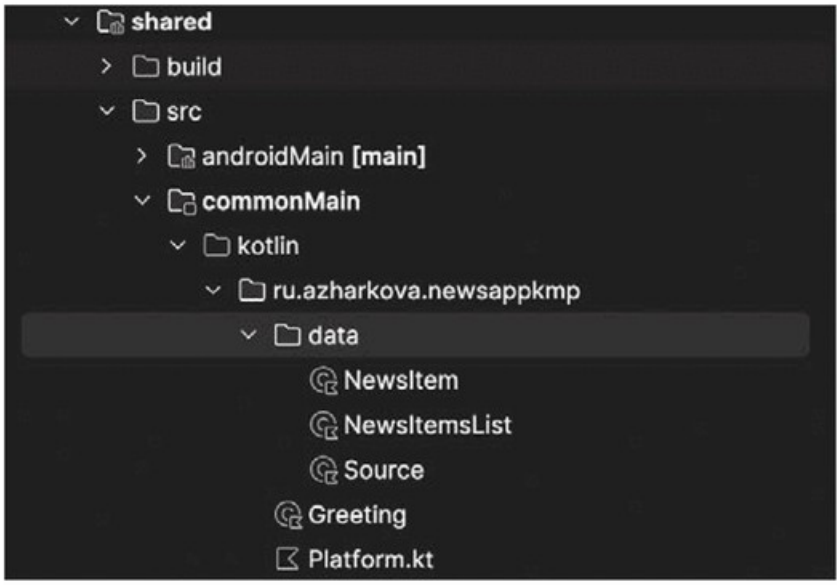


Рис. 2.3. Расположение структур данных

Код моделей приведен в листинге 2.1.

Листинг 2.1. Структуры данных для нашего приложения

```
// Источник новости
data class Source(
    val id: String,
    val name: String,
)

// Основная модель новости
data class NewsItem(
    val source: Source? = null,
    val author: String,
    val title: String,
    val description: String,
    val url: String? = null,
    val publishedAt: String? = null,
    val content: String,
    val urlToImage: String? = null
)

//Список новостей
data class NewsItemsList(
    val totalResults: Int,
    val articles: List<NewsItem>
)
```

Пояснение для разработчиков iOS. В Kotlin нет структур (struct, модель данных для хранения совокупности свойств по значению), поэтому для цели обработки и передачи данных используется специальный вид классов data class. Он легковесен и неизменяем (его атрибуты нельзя заменить после инициализации). Подробнее о различиях в Swift и Kotlin и работе с ними в KMP будет сказано в главе 3 про `expect/actual`. В главе 12 вы найдете шпаргалку по языкам.

Данные мы будем получать по сети. Значит, потребуется подключить десериализацию в приложение и добавить поддержку в наши data-классы. Дополнительные действия для поддержки десериализации и передачи данных между экранами и т. п. пока не прописываем. Все это мы сделаем в главе 5, где рассказывается о сетевом слое.

Переходим к сервисам, сетевому клиенту и хранилищу (рис. 2.4). Саму реализацию мы сейчас писать не будем, оставим для соответствующих глав. Наметим только структуру и связи.

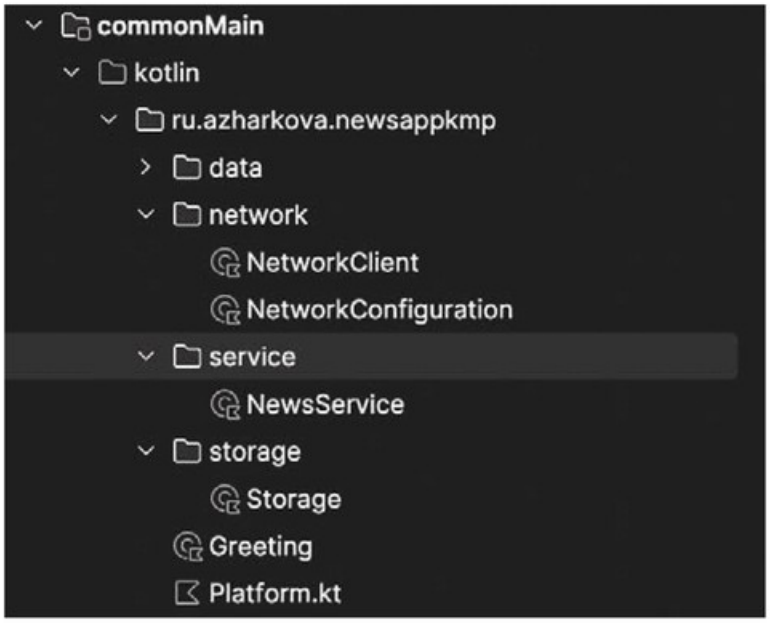


Рис. 2.4. Заготовки сервисов

Для работы с API newsapi.org нам понадобится сохранить базовый url хоста и ключ API. Поместим их в отдельный класс (листинг 2.2).

Листинг 2.2. Конфигурационные данные

```
class NetworkConfiguration {
    companion object {
        const val BASE_URL = "https://newsapi.org/v2/"
        const val API_KEY = "**********"
    }
}
```

В Kotlin нет статических методов и свойств. Использование companion object позволяет получать доступ к свойствам и константам класса без создания его инстанса.

В дальнейшем мы будем использовать механизм инжекции зависимостей DI для связей между компонентами бизнес-логики. Пока просто будем передавать зависимости через конструктор напрямую (листинг 2.3).

Листинг 2.3. Заготовка сетевого клиента

```
class NetworkClient constructor(
    private val networkConfiguration: NetworkConfiguration
) {

    fun loadData() : NewsItemsList? {
        // TODO: здесь будет реализация запроса к API
        return null
    }
}
```

Приводим этот пример, чтобы показать, как мы можем упрощать передачу зависимостей. В случае `NetworkConfiguration` мы получаем доступ к полям класса через Companion Object, поэтому можно убрать его из конструктора.

В сетевом клиенте пока оставим черновую реализацию-заглушку для будущих методов загрузки данных.

Добавим аналогичный класс-заготовку для хранилища (листинг 2.4).

Листинг 2.4. Заготовка для реализации Storage

```
class Storage  {

    fun saveData(newsItem: NewsItem) {
        // TODO: здесь будет реализация сохранения данных
    }
}
```

Теперь создадим сервис для работы с API новостей через сетевой клиент. Также заложим в нем возможность работы с хранилищем (листинг 2.5).

Листинг 2.5. Заготовка для реализации сервиса новостей

```
class NewsService
 constructor(
     private val networkClient: NetworkClient,
     private val storage: Storage
 ){

     fun loadData() {
         // TODO: здесь будет реализация запроса к API
     }
}
```

Предварительно можно убрать передачу хранилища в сервис новостей. Оно нам не понадобится до главы 8.

Хранилище, сетевой клиент и сервис новостей — это и будет наша общая бизнес-логика.

2.2. Архитектура кросс-платформенного приложения

Теперь подумаем, как использовать общий модуль бизнес-логики с нашими нативными приложениями. Согласно концепции KMP, в нативных приложениях мы реализуем то, что реализуется только нативным SDK и/или не поддерживается в общем модуле: это UI и работа с платформенными особенностями. Все остальное можем вынести в общий код:

- модели и структуры данных;
- сетевой слой и слой хранилища;
- сервисы/менеджеры/хелперы и прочие компоненты бизнес-логики, работающие с сетевым слоем и слоем хранилища.

Именно это сейчас и реализовано в нашем каркасе. Мы добавили data-классы и создали пустые классы для компонентов бизнес-логики, которые наполним логикой по мере освоения нашей дорожной карты. В целом мы уже можем подключать это к нашим нативным приложениям как внешний модуль библиотеки и использовать в коде.

Давайте разделим возможные варианты реализации архитектуры по уровням обобщения кода. Именно от этого аспекта и зависит итоговое решение:

- общая декларация, нативная реализация;
- общая бизнес-логика, нативная экранная архитектура;
- общая бизнес-логика, общая экранная архитектура;
- общая бизнес-логика, общая экранная архитектура и общий UI.

Начнем с первого уровня.

2.2.1. Нативное приложение и общий интерфейс

На рис. 2.5 изображены примерная иерархия и связи компонентов приложения KMP.

Business logic Common — это наш shared-модуль. iOS BL и Android BL — часть бизнес-логики, которая реализуется в нативном приложении. Давайте посмотрим, что это может быть за бизнес-логика и почему ее нельзя вынести в общий модуль.

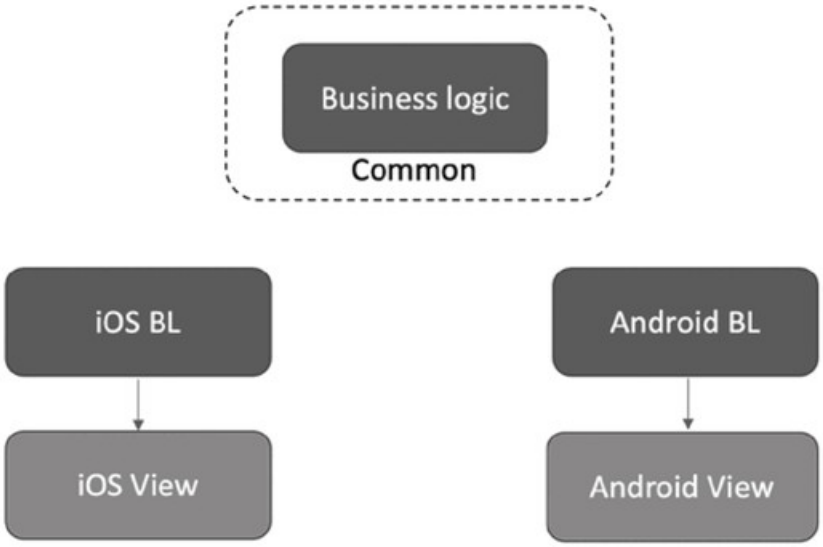


Рис. 2.5. Вариант реализации архитектуры

Возьмем пример, когда большая часть кода относится только к платформенной логике. Будьте готовы к глубокому погружению в задачу.

Обработка push-уведомлений

Для работы с push-уведомлениями на iOS используется механизм Apple Push Notification Service (APNs), работа идет напрямую через инфраструктуру Apple. Также есть своя строгая система авторизации и сертификации и свой центр UNUserNotificationCenter для управления уведомлениями. Для Android используется Firebase Cloud Messaging (FCM) как основной сервис уведомлений, а работа идет через Google Play Services. Также используется система обработки уведомлений FirebaseMessagingService. Это все нативная логика, работать с которой надо в нативных приложениях.

Мы можем реализовать работу с пуш-уведомлениями целиком и полностью в нативных приложениях. Но мы можем использовать и shared-модуль Kotlin Multiplatform для устранения избыточности в коде и выноса одинаковой логики в общий код.

Начнем с декларирования в commonMain интерфейса/протокола для нашего менеджера по работе с пушами. Это позволит нам управлять доступом пользователя кода к исходникам (листинг 2.6).

Листинг 2.6. Интерфейс для обработчика пуш-уведомлений

```
interface PushNotificationManager {
  suspend fun registerForPushNotifications(): PushNotificationResult
  suspend fun unregisterFromPushNotifications()
  suspend fun getToken(): PushNotificationResult
  fun setNotificationHandler(handler: (NotificationPayload) -> Unit)
}
```

Также создадим data-класс для `payload` данных и модель для разделения типов результата. Будем использовать sealed-класс с внутренними классами (листинг 2.7).

Листинг 2.7. Модель данных пуша

```
data class NotificationPayload(
        val title: String,
        val body: String,
        val data: Map<String, String>
)

// Результат
sealed class PushNotificationResult {
    data class Success(val token: String) : PushNotificationResult()
    data class Failure(val error: Throwable) : PushNotificationResult()
}
```

На iOS этот код трансформируется в класс с двумя внутренними классами (листинг 2.8).

Листинг 2.8. Преобразование Kotlin модели в Swift

```
open class PushNotificationResult : KotlinBase {
}

extension PushNotificationResult {

  public class Failure : PushNotificationResult {

    public init(error: KotlinThrowable)

    open var error: KotlinThrowable { get }
  }

  public class Success : PushNotificationResult {

    public init(token: String)

    open var token: String { get }
  }
}
```

Мы сможем вызывать этот функционал и со стороны iOS, прямо из нативного приложения.

В целом мы могли бы использовать и нативный для Kotlin тип `Result<T>`, тем более что аналогичный есть и в Swift, но он конвертируется в тип `Any` Swift, и данные могут потеряться. Поэтому мы создадим и применим свой класс-обертку. Подробнее о конвертации типов и конструкций читайте в главе об общем и нативном коде (глава 3). Далее займемся реализацией нашего менеджера уведомлений (листинг 2.9).

Листинг 2.9. Примерный код менеджера уведомлений

```
class NotificationManagerImpl(
    private val platformNotifier: PlatformNotifier
) : PushNotificationManager {

    private var notificationHandler: ((NotificationPayload) -> Unit)? = null

    override suspend fun registerForPushNotifications(): PushNotificationResult {
        return try {
            PushNotificationResult.Success(platformNotifier.register())
        } catch (e: Exception) {
            PushNotificationResult.Failure(e)
        }
    }

    override suspend fun unregisterFromPushNotifications() {
        platformNotifier.unregister()
    }

    override suspend fun getToken(): PushNotificationResult {
        return try {
            PushNotificationResult.Success(platformNotifier.getToken())
        } catch (e: Exception) {
            PushNotificationResult.Failure(e)
        }
    }

    override fun setNotificationHandler(handler: (NotificationPayload) -> Unit) {
        notificationHandler = handler
    }

    fun handleNotification(payload: NotificationPayload) {
        notificationHandler?.invoke(payload)
    }
}
```

Это примерный код обработки, который используется в таких случаях. Также в нем могут быть вызовы сетевого слоя для отправки токена на сервер или слоя хранилища для сохранения. По сути, этот менеджер является оберткой над типом `PlatformNotifier`. Этот класс нужен для подписки на уведомления, отписки, получения токена и его обработки (например, отправки на удаленный сервер). И этот `PlatformNotifier` как раз и относится к той части бизнес-логики, которую мы можем реализовать в нативных приложениях. Для унификации будем использовать `PlatformNotifier` как интерфейс (листинг 2.10).

Листинг 2.10. Интерфейс для оповещателя с платформы

```
interface PlatformNotifier {  suspend fun register():String  fun unregister()

  fun getToken():String
}
```

Для iOS у нас будет примерно такой код (листинг 2.11).

Листинг 2.11. Реализация на iOS

```
public class PlatformNotifierIOS: PlatformNotifier {


  public func register(completionHandler: @escaping (String?, (any Error)?) ->
Void) {
    UNUserNotificationCenter.current().requestAuthorization(options: [.alert,
.sound]) { granted, error in
      guard granted else { return }

      DispatchQueue.main.async {
        UIApplication.shared.registerForRemoteNotifications()
      }

      completionHandler(nil, error)
    }
  }

  public func unregister() {
    UNUserNotificationCenter.current().removeAllDeliveredNotifications()
  }
```

Здесь мы используем механизмы APNS для необходимых нам операций.

Такой менеджер нотификаций полезен только на стороне нативных приложений, поэтому мы его подключаем в соответствующий функционал именно в приложениях. Инициализируем менеджер в AppDelegate iOS-приложения и в качестве параметра передаем `PlatformNotifier` нужного типа (листинг 2.12).

Листинг 2.12. Подключение в AppDelegate

```
// В AppDelegate
class AppDelegate: UIResponder, UIApplicationDelegate {

  var notificationManager: NotificationManagerImpl?

  func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    // Инициализация менеджера
    notificationManager = NotificationManagerImpl(PlatformNotifierIOS())
    notificationManager?.setNotificationHandler { payload in
      // Обработка уведомлений
    }

    return true
  }
```

В блоке обработки уведомления можно также вызвать парсинг полученных из сообщения данных, сохранить промежуточную информацию или вызвать сетевой запрос.

На Android мы используем механизм `FirebaseMessaging` (листинг 2.13).

Листинг 2.13. Реализация PlatformNotifier в Android

```
class PlatformNotifierAndroid constructor() : PlatformNotifier {

  private val firebaseMessaging = FirebaseMessaging.getInstance()

  override suspend fun register(): String {
    return firebaseMessaging.token.await()
  }

  override fun unregister() {
    firebaseMessaging.deleteToken()
  }

  fun getToken(): String {
    return firebaseMessaging.token
  }
}
```

Инициализацию также реализуем на стороне нативного приложения в Application (листинг 2.14).

Листинг 2.14. Подключение на стороне Android

```
// В Application классе
class MyApplication : Application() {
  override fun onCreate() {
    super.onCreate()
    FirebaseApp.initializeApp(this)

    // Инициализация менеджера
    val notificationManager = NotificationManagerImpl(
      PlatformNotifierAndroid()
    )
    notificationManager.setNotificationHandler { payload ->
      // Обработка уведомлений
    }
  }
}
```

Таким образом, у нас есть бизнес-логика, которую мы сделали общей на стороне KMP, и бизнес-логика нативная в нативных приложениях.

2.2.2. Бизнес-логика экранного модуля

В предыдущем примере мы рассмотрели случай объединения весьма специфической платформенной и общей логики. Бизнес-логика клиента также включает в себя компонент, который является посредником между UI и основной моделью данных:

- `presenter` в паттерне MVP;
- `ViewModel` в паттерне MVVM;
- `Presenter-Interactor`/`Presenter-UseCase` в паттерне CleanArch/Viper.

При этом для приложений Android и iOS эти паттерны могут быть реализованы по-разному, в зависимости от выбранного стека для UI (SwiftUI vs UIKit для iOS, Compose vs xml для Android), вкусовых предпочтений, а также разного подхода к вопросу архитектуры команд, которые работают над проектом. Компоненты-посредники могут содержать также дополнительную логику в зависимости от потребности нативного приложения той или иной платформы.

Давайте рассмотрим варианты реализации такой архитектуры. Сами приложения при этом у нас будут полностью нативными. Отдельно от общего модуля реализуется UI для iOS и Android, а также архитектура нативных приложений. При таком подходе мы будем вызывать код, например, сетевого клиента или сервиса напрямую из архитектурного компонента экранного модуля нативного приложения:

```
shared/
    api/
        network/
    domain/
        models/
androidApp/
    presentation/
        android-specific/
iosApp/
    presentation/
        ios-specific/
```

В общем модуле у нас есть сервис, который помогает совершать сетевые запросы для контента определенного типа (листинг 2.15).

Листинг 2.15. Примерная реализация сервиса новостей

```
class NewsService(private val httpClient: NetworkClient) {

  suspend fun loadNews(): Result<NewsList> {
    return httpClient.request(URL)
  }

  companion object {
    val URL = "https://newsapi.org/v2/everything?q=science"
  }
}
```

На стороне Android мы можем использовать обычную `ViewModel` из библиотеки `Jetpack`. Также для этого класса в примере используется Hilt в качестве Dependency Injection (листинг 2.16).

Листинг 2.16. ViewModel Android

```
// AndroidViewModel.kt
@HiltViewModel
class AndroidViewModel
@Inject constructor() : ViewModel() {
  val newsService = DI.getNewsService

  fun fetchData() = viewModelScope.launch {
    val result = newsService.loadNews()
    // Обработка результата
  }
}
```

Если наш UI реализован с помощью классического xml, то методы `ViewModel` мы вызываем из `Activity` (листинг 2.17).

Листинг 2.17. Вызов со стороны Activity

```
// AndroidActivity.kt
class MainActivity : AppCompatActivity() {
  private val viewModel: AndroidViewModel by viewModels()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Android-специфичная логика
    viewModel.fetchData()
  }
}
```

Если UI реализован на Composable-функциях, то запрос будет идти, скорее всего, из блока `LaunchedEffect` (листинг 2.18).

Листинг 2.18. Пример вызова ViewModel в Composable

```
@Composable
fun NewsListScreen(viewModel: NewsViewModel) {
  LaunchedEffect(Unit) {
    viewModel.fetchData()
  }
  // Наполнение UI контентом из ViewModel
}
```

Для iOS мы тоже можем использовать `ViewModel`, но свою. Это будет класс, реализующий протокол `ObservableObject` для поддержания связи с UI, написанным на SwiftUI. Пример кода — в листинге 2.19.

Листинг 2.19. ViewModel на стороне iOS

```
import SwiftUI
import shared

class NewsListVm: ObservableObject {
```

```
  @Published var news: [NewsItem] = []
  private var service = NewsService()

  func getNews() {
    service.loadNews { [weak self] (data: NewsList), error in
      guard let self = self else {return}
      self.news = data.articles ?? []
    }
  }
}
```

UI для iOS у нас может быть реализован как на UIKit, так и на SwiftUI. Таким образом, у нас получается облегченный подход к использованию общего кода, при этом мы его дублируем. Если у нас нет явной необходимости использовать внутри классов некоторые сугубо нативные зависимости (как в данном случае механизм DI), которые можно подключить только в нативных приложениях, то стоит рассмотреть в качестве варианта вынос архитектуры в общий модуль.

2.2.3. Общий архитектурный посредник

Следующий уровень обобщения кода — использование общего архитектурного посредника (рис. 2.6).

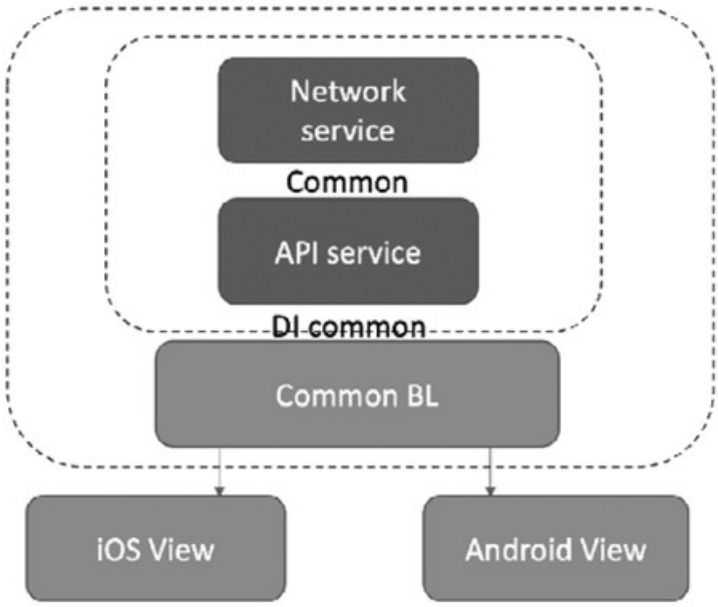


Рис. 2.6. Архитектура с общим посредником. Общая экранная архитектура

В общий модуль отправляется так называемая экранная бизнес-логика, она же посредник (`ViewModel`, `Presenter` и т. п.) из архитектурных паттернов MV(x). При правильной реализации (отсутствии прямых ссылок на UI и специфических библиотек для него) этот компонент легко обобщить и перенести в shared-модуль. Таким образом у нас возникает общая архитектура с общими точками

вызова наших сервисов. И речь идет не о каком-то одном файле, а о компонентах архитектуры каждого экранного модуля. Единую `ViewModel` на все экраны мы не рассматриваем.

Выбор такой архитектуры для экранов зависит от того, как у нас реализованы нативные приложения. Если для них выбраны DSL-фреймворки SwiftUI и Compose, то нам в любом случае потребуется реализовать специальные промежуточные модели состояния, которые и будут обращаться к нашему общему архитектурному компоненту. И этот архитектурный компонент уже может быть реализован по тому паттерну, который вы выберете.

Начнем с самого простого — презентера из MVP. Этот паттерн подразумевает, что связь между View и презентером, нативным и общим кодом происходит через специальные интерфейсы/протоколы, как в листинге 2.20.

Листинг 2.20. Интерфейсы для MVP

```
// Интерфейс презентера
interface Presenter {
  var view: View?

// Подключаемся к View
  fun attach(view: View)

  fun detach()
}

//Интерфейс для View
interface View {
  var presenter: Presenter?
}
```

View — это интерфейс для визуальной части. Если мы разрабатываем нативное приложение не на SwiftUI/Compose, то тогда реализовать этот интерфейс можно прямо в `UIViewController`/`Activity` (листинг 2.21).

Листинг 2.21. Подключение презентера к Activity

```
class MainActivity : ComponentActivity(), View {
// По-хорошему нужен DI и конфигуратор, но мы делаем лениво
  override val presenter: Presenter? by lazy { NewsPresenter() }

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Код по работе
  }

  override fun onResume() {
    super.onResume()
    presenter?.attach(this)
  }
}
```

Аналогичный презентер создадим для iOS (листинг 2.22).

Листинг 2.22. Подключение презентера к ViewController

```
class NewsListVC: UIViewController {

  // Тоже подключаем лениво
  private lazy var presenter: NewsPresenter? = {
    let _presenter = NewsPresenter()
    _presenter.attach (view: self)
    return _presenter
  }()

  // Код ViewController
}
```

Интерфейсы и презентер расположим в commonMain (листинг 2.23).

Листинг 2.23. Пример презентера

```
class NewsPresenter() : Presenter {
  override var view: View? = null

// Подключаем к View
  override fun attach(view: View) {}

  override fun detach() {}

}
```

Если же мы используем Compose/SwiftUI, то интерфейс View будет реализовывать компонент модели состояния (листинг 2.24).

Листинг 2.24. Модель состояния

```
class NewsListVM(items: List<NewsItem>?) : View {
    // На самом деле это не View, но ее модель состояния
    override val presenter: Presenter? by lazy {
        NewsPresenter().apply {
            attach(this@NewsListVM)
        }
    }

    fun loadNews() {}
}
```

Применим этот код для iOS и подключим презентер в модели состояния на ObservableObject (листинг 2.25).

Листинг 2.25. Модель состояния iOS

```
class NewsListVm: ObservableObject, View {

    lazy var presenter: NewsPresenter? = {
       let presenter = NewsPresenter()
```

```
        presenter.attach(view: self)
        return presenter
    }()
    @Published var news: [NewsItem] = []

}
```

В таком варианте мы из `ViewModel` Android/IOS вызываем общий презентер из commonMain.

В нашем приложении мы оптимизируем архитектуру немного по-другому. Так как у нас будет SwiftUI и Compose, семантически правильнее будет использовать `ViewModel` (рис. 2.7).

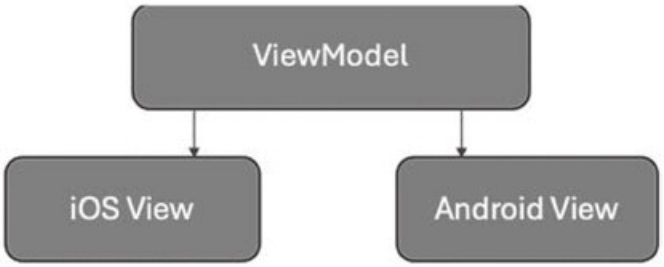


Рис. 2.7. Общая для экрана iOS и экрана Android ViewModel

Расположим `ViewModel` в commonMain:

```
open class BaseViewModel
```

Мы будем использовать базовую `ViewModel`, которую в следующих главах наполним логикой, в частности, для работы с многопоточностью.

Теперь добавим в commonMain каркас `ViewModel` для нашего экрана (листинг 2.26).

Листинг 2.26. NewsViewModel

```
class NewsViewModel() : BaseViewModel() {

  private val service = NewsService()
}
```

Такой компонент можно подключать на Android напрямую. Для iOS потребуется обращаться из модели состояния `ObservableObject` (листинг 2.27).

Листинг 2.27. Подключение ViewModel к модели состояния iOS

```
class NewsListVm: ObservableObject {

    @Published var news: [NewsItem] = []

    private lazy var vm: NewsViewModel? = {
```

```
// ViewModel
         let vm = NewsViewModel()
        vm?.newsFlow.collect(collector: itemsCollector,
                             completionHandler: {_ in})
         return vm
     }()
}
```

Мы начали с экрана списка новостей. Остальные модули доработаем позже.

Самый последний уровень обобщения кода с общим UI на Compose Multiplatform мы рассмотрим в главе 10.

2.3. UI нативных приложений

Изначально при создании приложения KMP мы получаем общий модуль и два нативных приложения: IOS и Android. Каждое из них можно реализовать как с помощью DSL-фреймворка (Compose или SwiftUI), так и с помощью классического UI-стека (xml или UIKit). Сейчас для нас UI не особо важен, поэтому выберем реализацию попроще и побыстрее: SwiftUI и Compose.

2.3.1. UI на SwiftUI

Как и Compose, SwiftUI — это декларативный фреймворк для разработки приложений. В отличие от Jetpack Compose, на верхнем уровне в SwiftUI декларативно все: экраны, контролы, навигация, стилизация.

Вместо контролов `UIView` и экранов `UIViewController` SwiftUI оперирует специальными структурами, реализующими протокол View. При этом контролы View являются высокоуровневыми обертками над UIKit-элементами. UI задается внутри свойства body конкретного View.

Стилизация реализуется с помощью применения модификаторов (modify) к конкретному View. Структуры не поддерживают наследование, поэтому новый тип View создается с помощью композиции (листинг 2.28).

Листинг 2.28. Пример создания View на основе Text с примененными свойствами

```
import SwiftUI

struct HeaderText: View {
    @State var text: String

    var body: some View {
        Text(text)
            .font(.system(size: 16))
            .bold()
    }
}
```

UI создается путем композиции и передачи декларативного блока выполнения для создания View внутри родительского View с помощью механизма ViewBuilder. И уже внутри в зависимости от типа, параметров и свойств (встроенных или кастомизированных) родительского контейнера строится сам UI.

Например, код в листинге 2.29 превратится в UI, как на рис. 2.8.



Рис. 2.8. Итоговый UI

Листинг 2.29. Фрагмент UI на SwiftUI

```
HStack {
            VStack(alignment: .leading, spacing: 10) {
                HeaderText(text: data.title ?? "")
                SubheaderText(text: data.description ?? "")
                SmallText(text: data.publishedAt?.formatToString
                        ("dd.MM.yyyy") ?? "")
            }
            .fixedSize(horizontal: false, vertical: true)

            ThumbImage(withURL: data.urlToImage ?? "")
        }
```

В SwiftUI View является функцией состояния. Поведение UI зависит от специальных переменных, которые называются обертками свойств (property wrappers).

Основные используемые property wrappers — это:

- `@State` — для локальных переменных;
- `@Binding` — аналог weak, для передачи ссылки на значение;
- `@EnvironmentObject` — для передачи объектов между `View`;
- `@ObjectBinding`, `@ObservableObject` — для отслеживания изменения свойств модели с помощью средств фреймворка `Combine`. Вместе с ними также используется `@Published` для маркировки свойства, на которое можно подписаться со стороны.

Любое изменение значения `propertyWrapper` приводит к перестройке всего View. Поэтому не храните ничего внутри `body`.

Подробнее о SwiftUI можно прочитать в документации: https://developer.apple.com/tutorials/swiftui.

Вернемся к нашей задаче.

Для iOS реализуем простой экран списка с элементом для строки (листинг 2.30).

Листинг 2.30. Код для элемента строки

```
struct NewsItemRow: View {
  // Локальная переменная
  @State var item: NewsItem

  var body: some View {
    // Горизонтальный стек
    HStack(alignment: .top) {
      // Изображение с загрузкой по url
      KFImage(url: URL(string: item.urlToImage ?? ""))
      // Вертикальный стек
      VStack(alignment: .leading, spacing: 7) {
        // Текстовый поля
        Text(item.title ?? "").lineLimit(4).textTitle()  // Модификаторы стиля
        Text(item.description ?? "").lineLimit(4).subtextTitle()
        Text(item.publishedAt?.formatToString("dd.MM.yyyy") ?? "").smallTitle()
      }
    }.background(Color.white)  // Заливка
  }
}
```

Наш элемент будет состоять из изображения слева и трех текстовых полей, расположенных вертикально справа от картинки (рис. 2.9).



Рис. 2.9. Пример готовой верстки элемента строки

Поскольку модель наших данных содержит поле для изображения, будем использовать библиотеку Kingfisher (https://github.com/onevcat/Kingfisher) для загрузки.

Сам экран списка будет иметь вид, как в листинге 2.31.

Листинг 2.31. View для экрана со списком

```
struct NewsListView: View{
// Слушаем модель состояния
    @ObservedObject var model = NewsListVm()

    var body: some View {
// Обертка для навигации
        NavigationView {
// Список элементов
            List(model.items, id: \.self) { item in
                NavigationLink(destination: NewsItemView(item: item)) {
                    NewsItemRow(item: item)
                }
            }.navigationBarTitle("News", displayMode: .inline)
        }
    }
}
```

Используем `NavigationView` для обертки в навигационный стек для навигации на экран единичной новости.

Добавим простой `List` в SwiftUI с указанием типа View для строки.

2.3.2. UI на Jetpack Compose

Jetpack Compose появился примерно в то же время, что и SwiftUI. Но, в отличие от декларативного iOS, здесь используются специальные функции-композиции, помечаемые аннотацией `@Composable`.

В остальном подход такой же: навигация, стилизация и собственно композиция UI реализуются декларативно. Для зависимости от моделей состояния могут использоваться специальные функции `remember`. Данные могут передаваться как через инициализатор, так и в качестве подписки на реактивное свойство из `ViewModel`.

Свойства задаются через специальные модификаторы.

Добавим `Composable` для строки списка (листинг 2.32).

Листинг 2.32. Composable для строки списка

```
@Composable
fun NewsListItemView(item: NewsItem, modifier: Modifier = Modifier) {
// Горизонтальный контейнер
    Row(
            modifier = modifier
                    .fillMaxWidth()
```

```
                    .padding(8.dp,8.dp),
            verticalAlignment = Alignment.Top,
            horizontalArrangement = Arrangement.SpaceAround,
    ) {
// Изображение с загрузкой по url
        Image(painter = rememberImagePainter(item.urlToImage.orEmpty()),
                contentDescription = "image",
                modifier = Modifier.width(120.dp).height(120.dp).
                           padding(0.dp, 0.dp, 8.dp, 0.dp))
// Вертикальный контейнер
        Column {
// Текст с настройками стиля
            Text(text = item.title.orEmpty(), style =
                        MaterialTheme.typography.titleMedium)
            Text(text = item.content.orEmpty(), style =
                        MaterialTheme.typography.titleSmall, maxLines = 3)
            Text(text = item.publishedAt.orEmpty(), style =
                        MaterialTheme.typography.bodyMedium)
        }
    }
}
```

Реализуем такой же UI, что и для iOS, но средствами Compose. Обратите внимание, что вы можете менять разметку на свое усмотрение и настраивать ее под себя.

Общий UI для списка будет иметь вид (листинг 2.33):

Листинг 2.33. Composable для списка

```
@Composable
fun NewsListView(items: List<NewsItem>) {
  LazyColumn(
    contentPadding = // 1.
      PaddingValues(horizontal = 16.dp, vertical = 8.dp)
  ) {
    items(items) {
// Composable элемента списка
      NewsListItemView(it)
    }
  }
}
```

Выглядит очень похоже на код для iOS. `LazyColumn` и `items` позволяют нам создать последовательность элементов в виде списка, как `List` в SwiftUI.

Composable, на который подписаны контролы, перестраивается при изменении реактивных свойств, поэтому объявление и создание `ViewModel` в его скоупе приведет к постоянному пересозданию и сбросу состояния. Чтобы избежать такого поведения, будем передавать `ViewModel` для Android в Composable из `Activity`. Добавляем такой параметр в конструктор (листинг 2.34).

Листинг 2.34. Передача ViewModel в Composable через конструктор

```
@Composable
fun NewsListScreen(viewModel: NewsViewModel) {
  LaunchedEffect(Unit) {
    viewModel.loadNews()
  }

  NewsListView(emptyList())
}
```

ПОДВЕДЕМ ИТОГИ

- ✓ Мы подготовили каркас приложения.
- ✓ Добавили UI на Compose и SwiftUI.
- ✓ Выбрали способ работы с shared-модулем и подготовили архитектуру.

Со следующей главы мы начинаем погружение в детали Kotlin Multiplatform. Вы можете наполнять каркас приложения постепенно по мере чтения или перейти сразу к главе 6 «Собираем приложение по частям».

ГЛАВА 3

Expect/actual. Общее и нативное

В этой главе рассмотрим:

- что делать общим, а что нативным;
- чем различаются Kotlin JVM для Android и Kotlin Native для iOS;
- каковы особенности реализации кода для Kotlin Native;
- как обобщить код для решения нативных задач с помощью интерфейсов;
- как работать с `expect/actual`;
- какие нюансы компиляции shared-кода для Kotlin Native прячутся под капотом;
- как происходит прямое преобразование Kotlin в Swift и каковы особенности его текущей реализации.

Kotlin Multiplatform позволяет упростить и ускорить разработку за счет реализации модуля общего кода shared. По рекомендации разработчиков SDK в shared-модуле следует размещать код, который не относится к тому, что сильно зависит от платформы, то есть всё, кроме UI и, например, работы с сенсорами аппарата. К этим общим данным относятся:

- модели данных (структуры данных и бизнес-логика);
- слой по работе с сетью;
- слой по работе с хранилищами данных;
- архитектура и т. п.

Для многих из этих задач уже есть готовые библиотеки и решения как от компании JetBrains, так и от других разработчиков.

Несмотря на то что большая часть кода в shared-модуле будет одинаковой для всех таргетов, некоторый функционал требует реализации с учетом особенностей каждой платформы. Как мы помним из предыдущих глав, это обусловлено тем, что shared-модуль у нас не сплошной и разбивается на подмодули для каждого таргета, причем в каждом из этих подмодулей для каждого из таргетов используется своя особая версия Kotlin (рис. 3.1).

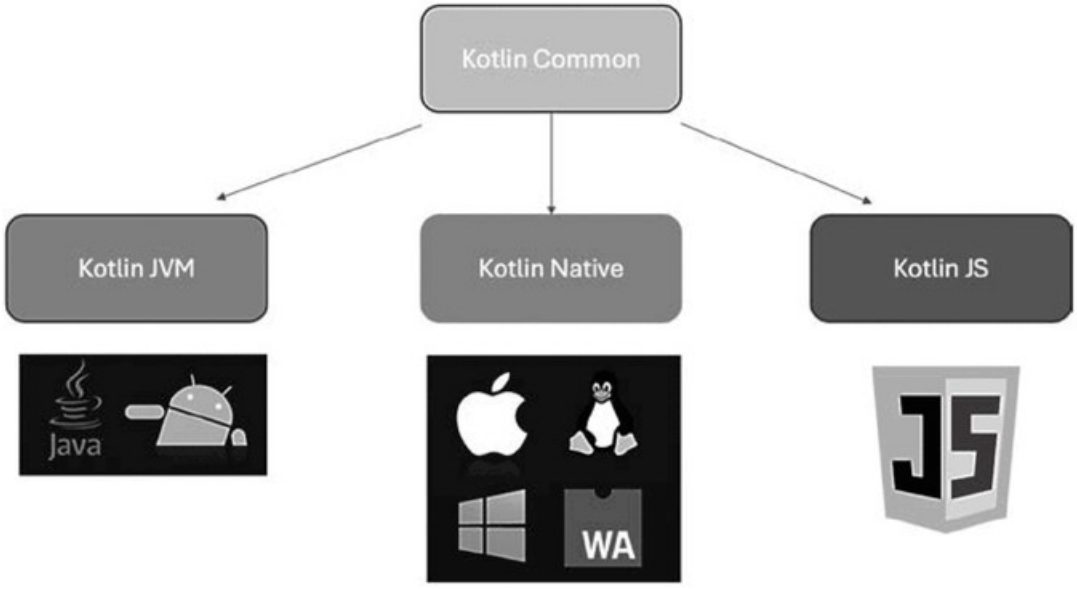


Рис. 3.1. Версии Kotlin для таргетов

Kotlin Common — это общая версия для commonMain; Kotlin JVM используется для Java и Android (androidMain), Kotlin Native — для iOS (iOSMain), Linux, Mac OS, Kotlin JS/Wasm — для веб.

Kotlin Multiplatform поддерживает возможность задекларировать что-нибудь в общем коде (commonMain) как единую общую точку и по-разному реализовать в платформенных частях shared-модуля (iOSMain, androidMain и т. п.) с помощью механизма `expect/actual` (рис. 3.2).

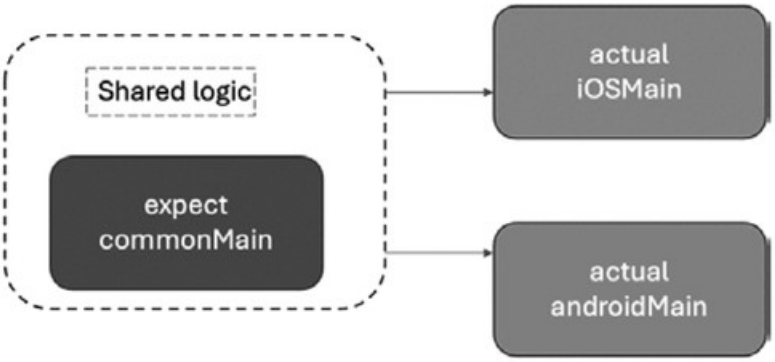


Рис. 3.2. Соотношение expect/actual

В главе 1, посвященной настройке базового проекта и приложения Kotlin Multiplatform, мы рассмотрели встроенный пример с выводом версии ОС каждого таргета. Теперь мы обсудим особенности работы `expect/actual` и общего/нативного кода более детально.

Мы не будем подробно рассматривать нюансы интеграции между Kotlin Common и Kotlin JVM. Для Android-таргета Kotlin является нативным, и преобразование между Common- и JVM-версиями происходит бесшовно и беспроблемно. Единственное ограничение в данном случае: в commonMain нельзя использовать JVM напрямую, если Android не является единственным таргетом вашего приложения. Но в этом случае теряет смысл работа с кросс-платформой, так как правильнее будет делать полностью нативное приложение. JVM не поддерживается Kotlin Native, который используется для iOS-таргета и десктопов.

Работа с iOS-таргетом и десктопами является самой сложной частью разработки приложения KMP за счет различий между iOS SDK, Android SDK, Kotlin и Swift. Поэтому мы и уделим основное внимание особенностям и специфике работы с Kotlin Native. Начнем с некоторой теории по интеграции разных версий Kotlin в единый shared-модуль и поддержке/преобразовании синтаксиса Swift/Objective-C в Kotlin и обратно.

3.1. Интеропедия. Библиотека преобразований Kotlin в Objective-C/Swift

Kotlin/Native поддерживает двунаправленную совместимость (interopability, далее интероп) с Objective-C. Это означает, что и портированные по умолчанию библиотеки, и фреймворки Objective-C могут использоваться в Kotlin-коде (с Kotlin/Native в iOSMain), и Kotlin-код будет преобразован в код Objective-C. Чтобы узнать, как работает этот функционал внутри, перейдите к разделу со звездочкой (3.3). Пока мы сосредоточимся на практической стороне вопроса.

Несмотря на то что внешне синтаксис языков Kotlin и Swift похож и многие вещи сейчас конвертируются под капотом практически бесшовно для пользователя SDK KMP, есть и определенные различия, которые могут потребовать доработки или обходного решения. Все-таки iOS SDK и Android SDK — явно не одно и то же, как и сами платформы.

Для корректной интеграции нативного iOS-приложения с Kotlin-кодом shared-модуля нам нужно знать и уметь пользоваться правилами преобразования синтаксических конструкций Kotlin в Objective-C/Swift.

Основные правила интеропа изложены в специальной таблице совместимости на сайте с документацией по Kotlin Multiplatform https://github.com/kotlin-hands-on/kotlin-swift-interopedia (табл. 3.1).

Таблица 3.1

Kotlin	Swift	Objective-C
class	class	@interface
interface	protocol	@protocol
constructor/create	Initializer	Initializer
Property	Property	Property
Method	Method	Method
enum class	class	@interface
suspend->	completionHandler:/async	completionHandler:
@Throws fun	throws	error:(NSError**)error
Extension	Extension	Category member
companion member <-	Class method or property	Class method or property
null	nil	Nil
Singleton	shared or companion property	shared or companion property
Primitive type	Primitive type/NSNumber	
Unit return type	Void	Void
String	String	NSString
String	NSMutableString	NSMutableString
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
MutableSet	NSMutableSet	NSMutableSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary
Function type	Function type	Block pointer type
Inline classes	Unsupported	Unsupported

Также в этой таблице приводится информация о конвертации типов Kotlin — Swift — Objective-C и о преобразовании Any и его методов (табл. 3.2).

Таблица 3.2

Kotlin	Swift	Objective-C
equals()	isEquals(_:)	isEquals:
hashCode()	hash	hash
toString()	description	description

3.1.1. Базовые примеры конвертации

Давайте пройдемся по некоторым из основных примеров конвертации из Kotlin в Swift.

Для Swift-версий Kotlin-типов используется префикс `Kotlin-` (листинг 3.1). В этом случае создается специальный тип `KotlinInt`, наследующий `Kotlin Number`. Это портированная версия типа `Int32` из Kotlin.

Листинг 3.1. Тип KotlinInt

```
open class KotlinInt : KotlinNumber {

    public init(value: Int32)

    public convenience init(int value: Int32)
}
```

Kotlin-классы являются по умолчанию финальными, а в Swift все классы, наоборот, — открытыми для наследования, то есть `open`.

В Kotlin используется класс `Any` в качестве базового для всех типов. Для Swift для него генерируется класс `KotlinBase`, имплементирующий интерфейсы/протоколы `NSObject` и `NSCopying` (листинг 3.2).

Листинг 3.2. Базовый класс KotlinBase

```
open class KotlinBase : NSObject {

    open class func initialize()
}

extension KotlinBase : NSCopying {
}
```

Правил довольно много, и они затрагивают практически все основные синтаксические единицы. Подробнее вы можете прочесть в официальной документации: https://kotlinlang.org/docs/native-objc-interop.html.

Обратите внимание, что описываемое состояние учитывает актуальную версию Kotlin. Если вы имеете дело с более ранней версией, то можете столкнуться с другими ограничениями, которые решены, например, в более поздней версии Kotlin.

На данный момент большинство синтаксических конструкций автоматически конвертируются между Swift/Objective-C и Kotlin, поэтому имеет смысл обновляться до самой актуальной версии языка, чтобы не тратить время на ненужные обходные решения. Например, рассмотрим конвертацию sealed-классов из предыдущей главы. В shared-модуле в commonMain мы создали класс для обработки результатов от нашего пуш-менеджера (листинг 3.3).

Листинг 3.3. Sealed-класс Kotlin

```
sealed class PushNotificationResult {
    data class Success(val token: String) : PushNotificationResult()
    data class Failure(val error: Throwable) : PushNotificationResult()
}
```

Kotlin/Native сгенерирует для Swift класс с двумя внутренними классами (листинг 3.4).

Листинг 3.4. Конвертация sealed-класса в Swift

```
open class PushNotificationResult : KotlinBase {
}

extension PushNotificationResult {

    public class Failure : PushNotificationResult {

        public init(error: KotlinThrowable)

        open func doCopy(error: KotlinThrowable) ->
                        PushNotificationResult.Failure

        open func isEqual(_ other: Any?) -> Bool

        open func hash() -> UInt

        open func description() -> String

        open var error: KotlinThrowable { get }
    }

    public class Success : PushNotificationResult {

        public init(token: String)

        open func doCopy(token: String) -> PushNotificationResult.Success

        open func isEqual(_ other: Any?) -> Bool
```

```
        open func hash() -> UInt

        open func description() -> String

        open var token: String { get }
    }
}
```

Data-классы Kotlin являются наследниками базового типа `Any`, который конвертируется в KotlinBase.

Обратите внимание, что кроме свойств, которые мы задекларировали для внутренних классов, сгенерировались и базовые методы data-классов. Например, data-классы поддерживают копирование, поэтому создается метод `doCopy(:_)`. Тип `KotlinThrowable`, как и `Error` в Swift, используется для ошибок, но с той лишь разницей, что `Error` — это протокол (интерфейс), а `KotlinThrowable` — класс.

Обработать результат в нативном iOS-приложении мы можем вот так (листинг 3.5):

Листинг 3.5. Обработка sealed-класса в iOS

```
func handlePushNotificationResult(result: PushNotificationResult) {
        switch result {
        case let success as PushNotificationResult.Success:
            let token = success.token
            print("Success: \(token)")

        case let failure as PushNotificationResult.Failure:
            let error = failure.error
            print("Failure: \(error)")

        default:
            print("Unknown result")
        }
    }
```

В какой-то мере это похоже на `enum`.

При этом сами `enum` Kotlin преобразуются в Objective-C в интерфейсы, а в Swift — в классы (листинг 3.6).

Листинг 3.6. Пример enum

```
// Kotlin
enum class Colors {
    RED, GREEN, BLUE
}

// Swift
Colors.red
Colors.green
Colors.blue
```

Способ обращения к ним в этом случае выглядит практически идентично нативному Swift enum, как в листинге 3.7.

Листинг 3.7. Switch-case обработка enum в Swift

```
switch color {
    case .red: print("It's red")
    case .green: print("It's green")
    case .blue: print("It's blue")
    default: fatalError("No such color")
}
```

В Kotlin часто используется декларация функций и свойств как глобальных на верхнем уровне (листинг 3.8).

Листинг 3.8. Глобальная функция в Kotlin

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

Но при преобразовании в Swift-код для таких деклараций генерируются классы с названием исходного файла и суффикса kt:

```
MyLibraryUtilsKt.foo()
```

И обращаться к таким функциям и свойствам нужно именно как к статическим функциям и свойствам данного класса. Это очень важно, иначе вы потратите много времени на поиски входных точек вашего кода и причин, почему они не отображаются (якобы) на iOS.

Также стоит отметить синглтоны и статические свойства/функции. В Kotlin нет статических типов, там используется специальный объект companion object, обращение к которому равносильно обращению к свойствам/методам класса (листинг 3.9).

Листинг 3.9. Синглтоны в Kotlin

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

Для создания объекта-синглтона используется специальный object. На Swift появляется промежуточное статическое свойство shared, через которое мы можем обратиться к companion (листинг 3.10).

Листинг 3.10. Доступ к синглтону Kotlin со стороны Swift

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

Если говорить о многопоточности, то сейчас `suspend`-методы корутин превращаются либо в `async/await`, либо в метод с блоком обратного вызова при завершении `completionHandler`. Подробнее о корутинах и их преобразовании см. в главе 4 про многопоточность. В целом интеграция Kotlin в Swift не требует сейчас почти никаких дополнительных действий от разработчика и пользователя SDK. Практически все интуитивно понятно, за исключением некоторых ограничений, о которых мы поговорим ниже.

3.1.2. Ограничения конвертации Kotlin в Swift

Основными актуальными ограничениями на данный момент являются:

1. Поддержка inline classes. Этот механизм обеспечивает встраивание кода класса (кода реализации) в место вызова. Очень специфический функционал сугубо для Kotlin. В Swift поддерживается встраивание с помощью `@inlinable`, но не сами inline-классы.
2. Наследование классов Kotlin в нативном iOS-приложении Swift- и Objective-C-классами. В данный момент не поддерживается корректно. Но имплементировать Kotlin-интерфейсы вполне возможно, так как они конвертируются в протоколы.
3. Наследование нефинальными (open) Kotlin-классами классов Swift не поддерживается. Этот кейс актуален для реализации кода внутри iOSMain на Kotlin Native.

   Однако мы вполне можем наследовать и переопределять классы Swift финальными классами Kotlin. Для этого надо будет помечать переопределяемые методы с помощью ключевого слова `override`. В случае, если надо переопределить инициализатор/конструктор, используйте ключевое слово (аннотацию) `@OverrideInit` (листинг 3.11).

   Листинг 3.11. Пример @OverrideInit для переопределения инициализатора

   ```
   class ViewController : UIViewController {
       @OverrideInit constructor(coder: NSCoder) : super(coder)

       ...
   }
   ```

   При переопределении конструктора потребуется указывать те же типы и параметры, что в исходном конструкторе/инициализаторе.

Чтобы компилятор игнорировал конфликтующие переопределения, необходимо пометить Kotlin-класс аннотацией `@ObjCSignatureOverride`.

4. Наследование и переопределение типов коллекций Kotlin: Set, List, Map. В данный момент не поддерживается. Коллекции Kotlin имеют во многом аналоги Swift и конвертируются в подходящие типы коллекций Swift без особых проблем. Рекомендуется использовать готовые существующие типы.
5. Поддержка Generics. И в Swift, и в Kotlin этот механизм есть, но его работа имеет свои особенности. Например, в Swift нет поддержки generic-протоколов, в отличие от Kotlin. Также из-за различий в работе с типами Kotlin и Kotlin Native может теряться информация.

   Kotlin Native работает с `ObjCClass` вместо `KClass` (листинг 3.12).

Листинг 3.12. Фрагмент кода преобразования типов ObjCClass в Kotlin Native

```
package kotlinx.cinterop
  import kotlin.native.*
  import kotlin.native.internal.*

  interface ObjCObject
  interface ObjCClass : ObjCObject
  interface ObjCClassOf<T : ObjCObject> : ObjCClass
// TODO: T should be added to ObjCClass and all meta-classes instead.
  typealias ObjCObjectMeta = ObjCClass

  interface ObjCProtocol : ObjCObject
```

Кроме того, типы для Kotlin и Swift будут называться по-разному. Вызываемые из shared-модуля типы будут иметь префикс с названием этого модуля (листинг 3.13).

Листинг 3.13. Фрагмент заголовочного файла модуля shared с конвертацией типов на Objective-C

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Greeting")))
@interface SharedGreeting : SharedBase
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__
((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
message="use object initializers instead")));
- (NSString *)greet __attribute__((swift_name("greet()")));
@end

__attribute__((swift_name("Platform")))
@protocol SharedPlatform
@required
@property (readonly) NSString *name __attribute__((swift_name("name")));
@end
```

Поэтому при попытке инициализировать тип по названию (`KClass`) мы получим `null` (листинг 3.14).

Листинг 3.14. Получение типа KClass по названию типа в Swift

```
fun<T:Any> createType(clazz: String):KClass<*>? {
  val objCClass = NSClassFromString(clazz)
  if (objCClass != null) {
    return getOriginalKotlinClass(objCClass)
  }
  return null // no type found
}
```

Правильно будет использовать передачу `ObjCClass` со стороны нативного iOS-приложения (листинг 3.15).

Листинг 3.15. Передача ObjCClass

```
fun<T:Any> createType(clazz: ObjCClass):KClass<*>? {
  return getOriginalKotlinClass(clazz)
}
```

Если мы решаем задачу получения существующего инстанса того или иного типа из общего сервиса `DependencyInjection` или создания нового инстанса с помощью некоторой фабрики, то для этого потребуется создать специальный метод в iOSMain как расширение данного функционала, уже реализованного в commonMain (листинг 3.16).

Листинг 3.16. Метод расширения для фабрики

```
fun<T:Any> SomeFactory.resolve(clazz: ObjCClass):T? {
    val kClass = createType<T>(clazz)
    if (kClass != null) {
        return getData(kClass)
    }
    return null
}
```

6. Аннотации Kotlin. Не поддерживаются на стороне Swift/Objective-C.
7. Абстрактные классы Kotlin. Не поддерживаются на стороне Swift/Objective-C.
8. Название классов и протоколов/интерфейсов. На стороне iOS при преобразовании интерфейса Kotlin в Swift-протокол его имя приобретает префикс, состоящий из названия модуля, а также пакетов (package). Например, `Kotlinx_coroutines_coreFlowCollector` или `SharedNetworkClient`.

Подробнее о кейсах и нюансах преобразования можно прочитать здесь: https://github.com/kotlin-hands-on/kotlin-swift-interopedia.

Перейдем теперь собственно к разбору механизма `expect/actual` и особенностям его использования.

3.2. Интерфейс или expect/actual

Механизм `expect/actual` отлично подходит для бесшовного соединения реализаций под разные платформы. Главное использовать его к месту и не злоупотреблять. Кроме инструкций `expect/actual` для поддержки разных реализаций в shared-модуле (и не только) можно применять стандартные интерфейсы. Разберем, какой подход и в каких случаях стоит использовать.

3.2.1. Приложение с нативным UI и общим интерфейсом для View

Начнем с простого. Представим, что у нас есть нативные приложения Android и iOS, у которых экраны реализованы с разной разметкой и на разных технологиях: Compose и SwiftUI. И мы хотим со стороны нашего общего модуля управлять, например, показом некоторых данных, событиями или даже задать общий стиль. `AndroidMain` умеет работать с Compose, iOSMain со SwiftUI. Нам поможет реализация общего интерфейса для View (листинг 3.17).

Листинг 3.17. Общий интерфейс для View

```
interface ISimpleColorView {

  var presenter: IColorScreenPresenter?

  fun showColor(color: Colors)
}
```

Также добавим сюда ссылку на интерфейс общего презентера (листинг 3.18).

Листинг 3.18. Общий презентер и реализация

```
class ColorScreenPresenter : IColorScreenPresenter{

  override var colorView: ISimpleColorView? = null

  // Код какой-то…

  fun showColor(color: Colors) {
    colorView?.showColor(color)
  }
}

interface IColorScreenPresenter {
  var colorView: ISimpleColorView?
}
```

Презентер мы реализуем в commonMain для единого и общего управления.

Сами цвета будем передавать из `enum Colors` (листинг 3.19).

Листинг 3.19. Enum цветов

```
enum class Colors {
    BLUE, RED, GREEN
}
```

А вот маппинг значений `Colors` в цвета придется делать в нативных приложениях. При попытке реализации в iOSMain/androidMain мы получим предложение установить библиотеку для работы с Compose UI для Kotlin Multiplatform (рис. 3.3). Сейчас мы этот случай не рассматриваем. Также данное решение не подходит для случаев, когда у нас строгое ограничение по используемым технологиям реализации экранов приложений именно нативно.

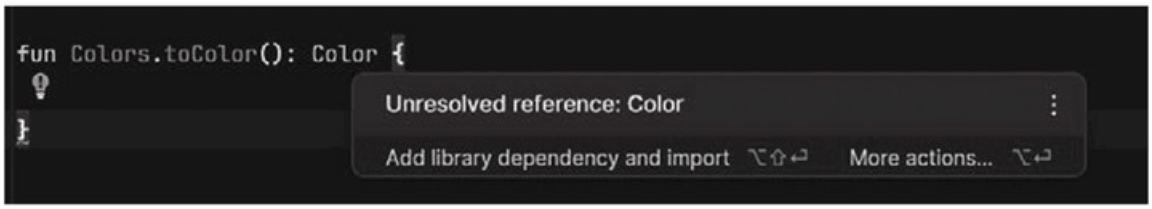


Рис. 3.3. Отсутствие общих ресурсов в приложении KMP

На стороне iOS реализуем свое расширение по преобразованию в `Color` для SwiftUI в нативном приложении (листинг 3.20).

Листинг 3.20. Расширение для цвета на стороне iOS

```
import shared
import SwiftUI

extension Colors {

    func toColor() -> Color {
        switch self {
        case .blue:
            return .blue
        case .red:
            return .red
        case .green:
            return .green
        default:
            return .clear
        }
    }
}
```

Аналогично реализуем свое расширение для Android в androidMain (листинг 3.21).

Листинг 3.21. Расширение для работы с цветом в Android

```
import androidx.compose.ui.graphics.Color
import ru.azharkova.colorchanger.data.Colors

fun Colors.toColor(): Color {
  return when(this) {
    Colors.BLUE -> Color.Blue
    Colors.RED -> Color.Red
    Colors.GREEN -> Color.Green
  }
}
```

Теперь время добавить интерфейс/протокол для экрана в нативных приложениях. И здесь мы сразу столкнемся с ограничением со стороны Swift (рис. 3.4).

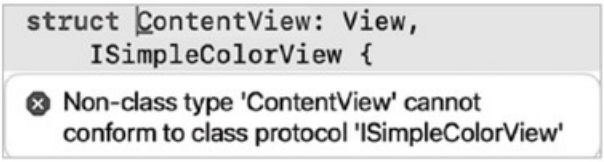


Рис. 3.4. Ограничения структур и имплементации общих протоколов в Swift

Мы не можем реализовать протокол структурой, потому что автоматически он считается протоколом, поддерживающим только классы.

Когда мы разрабатывали каркас учебного приложения и подбирали архитектуру, то проговорили сразу, что для декларативных фреймворков нам потребуется модель состояния для подключения бизнес-логики или ее компонентов и что сделать это напрямую к View получится не всегда. Для этого случая мы получили сейчас обоснование.

На стороне Android у нас Composable-функция, поэтому реализовать напрямую интерфейс View не получится.

Решение: будем использовать промежуточные классы состояний.

На iOS используем протокол `ObservableObject` SwiftUI (листинг 3.22).

Листинг 3.22. Прокси-модель состояния на iOS

```
class ContentViewModel: ObservableObject, ISimpleColorView {

    var presenter: (any IColorScreenPresenter)?

    @Published var color: Color = .clear

    func setup() {
        self.presenter = ColorScreenPresenter()
        self.presenter?.colorView = self
    }
```

```
    func showColor(color: Colors) {
        self.color = color.toColor()
    }
}
```

Генерируемый цвет сохраним в свойство `color`, на которое подпишемся со стороны View.

Связь между объектом и презентером настроим в методе `setup`, который будем вызывать в исходном View.

Подключим как `ObservedObject` к View (листинг 3.23).

Листинг 3.23. Подключение прокси-модели к View

```
struct ContentView: View {
    @ObservedObject var viewModel: ContentViewModel = .init()

           var body: some View {
        VStack(alignment: .center) {
            Text("Press for color")
            Button(action: {
// Действие по кнопке
                self.viewModel.presenter?.randomizeColor()
            }) {
                Text("Change color")
            }
        }.padding(10)
            .frame(
                  minWidth: 0,
                  maxWidth: .infinity,
                  minHeight: 0,
                  maxHeight: .infinity,
                  alignment: .center
                )
                      // Цвет из VM
            .background(viewModel.color)
            .onAppear {
                viewModel.setup()
            }
                  }
```

Наш View состоит из вертикального стека с текстом и кнопкой, по нажатии на которую мы будем вызывать действие.

Инициализацию можно подключить в `onAppear` `View` или при инициализации, так как сложной и тяжелой логики здесь нет.

Теперь в модификаторе `background` настроим зависимость фона от цвета `color`.

Саму смену цвета реализуем по кнопке. Для простоты решения добавим рандомизированную генерацию (листинг 3.24).

Листинг 3.24. Генерация цвета в презентере

```
class ColorScreenPresenter : IColorScreenPresenter{
    // Код какой-то

    override fun randomizeColor() {
        val colorCode = (0..2).random()
        val color = Colors.entries.find { it.color == colorCode}
        showColor(color ?: Colors.BLUE)
    }
}

interface IColorScreenPresenter {
    var colorView: ISimpleColorView?

    fun randomizeColor()
}
```

Теперь добавим прокси-модель состояния на стороне Android и реализуем интерфейс View (листинг 3.25).

Листинг 3.25. Модель состояния на стороне Android

```
class ColorVM (): ISimpleColorView {

  val color: MutableStateFlow<Color?> = MutableStateFlow(null)
  override var presenter: IColorScreenPresenter? = null

  fun setup() {
    this.presenter = ColorScreenPresenter()
    this.presenter?.colorView = this
  }

  override fun showColor(color: Colors) {
    this.color.tryEmit(color.toColor())
  }
}
```

Цвет мы будем сохранять в поле типа Kotlin `StateFlow`, чтобы отслеживать его изменения (листинг 3.26).

Листинг 3.26. Подключение и вызов модели состояния из Composable

```
@Composable
fun ColorView(vm: ColorVM, modifier: Modifier = Modifier) {
  val color = vm.color.collectAsState()

  LaunchedEffect(Unit) {
    vm.setup()
  }

// Фон из VM
```

```
  Column(modifier = modifier.fillMaxSize().background(color.value ?:
Color.White)) {
    Text(text = "Press button to change color")
    Button(onClick = {
// Действие по кнопке
      vm.presenter?.randomizeColor()
    }) {
      Text(text = "Change color")
    }
  }
}
```

Как и в Swift, наш экран состоит из текста и кнопки. Кликом по кнопке мы генерируем новое значение для фона экрана.

Вызов инициализации перенесем на событие запуска `LaunchedEffect` нашего `Composable`. Подписку на цвет реализуем с помощью расширения `collectAsState`. Также будем использовать его значение в модификаторе фона.

Проверим результат (рис. 3.5).



Рис. 3.5. Результат работы

Промежуточные выводы

Изначально кейсы работы с UI Android и iOS в KMP не имеют общего решения. iOSMain или androidMain не содержат нужных нам библиотек по работе

с UI, и если для Android-таргета мы можем добавить нужную зависимость, то для iOS так сделать не получится. Следовательно, полная реализация в общем модуле не имеет смысла.

Однако можно заложить единый алгоритм или шаблон работы с ними через общий интерфейс в shared-модуле.

3.2.2. Интерфейс vs. expect/actual

Теперь возьмем кейс, для которого разместить все решение в shared-модуле в целом возможно. Представим, что нам необходимо реализовать в общей части логику работы с акселерометром, но с учетом особенностей Android и iOS. Для Android этот механизм использует `SensorManager` и зависит от контекста `applicationContext` (листинг 3.27).

Листинг 3.27. Accelerometer для Android

```
class Accelerometer constructor(
        private val listener: (x: Float, y: Float, z: Float) -> Unit
) {
               // Зависит от контекста applicationContext
        private var sensorManager: SensorManager? = null
        private var accelerometer: Sensor? = null

        private var applicationContext: Context? = null
        private val sensorEventListener = object : SensorEventListener {
            override fun onSensorChanged(event: SensorEvent?) {
                if (event?.sensor?.type == Sensor.TYPE_ACCELEROMETER) {
                    val x = event.values[0]
                    val y = event.values[1]
                    val z = event.values[2]
                    listener(x, y, z)
                }
            }

            override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {}
        }

               // Подключаем контекст
        fun setupContext(applicationContext: Context?) {
            this.applicationContext = applicationContext
        }

        fun startListening() {
            val context = applicationContext
                    ?: throw IllegalStateException("Application context is
                                                    not available")
                                // Инициализируем через службы из контекста
            sensorManager = context.getSystemService(Context.SENSOR_SERVICE)
                            as SensorManager
```

```
            accelerometer = sensorManager?.getDefaultSensor(Sensor.
                            TYPE_ACCELEROMETER)
            if (accelerometer != null) {
                sensorManager?.registerListener(
                        sensorEventListener,
                        accelerometer,
                        SensorManager.SENSOR_DELAY_NORMAL
                )
            } else {
                throw IllegalStateException("Accelerometer sensor is
                                            not available")
            }
        }

        fun stopListening() {
            sensorManager?.unregisterListener(sensorEventListener)
        }
}
```

Для iOS мы используем CoreMotion `CMMotionManager` (листинг 3.28).

Листинг 3.28. Accelerometer для iOS

```
class Accelerometer constructor(
        private val listener: (x: Float, y: Float, z: Float) -> Unit
) {
    private val motionManager = CMMotionManager()

    @OptIn(ExperimentalForeignApi::class)
    fun startListening() {
        if (motionManager.accelerometerAvailable) {
            motionManager.accelerometerUpdateInterval = 0.1 // 10 Hz
            motionManager.startAccelerometerUpdatesToQueue(
                    NSOperationQueue.mainQueue()) { data, _ ->
                val acceleration: CValue<CMAcceleration> = (data
                                  as CMAccelerometerData).acceleration
                acceleration.useContents {
                    listener(this.x.toFloat(),
                            this.y.toFloat(), this.z.toFloat())
                }
            } else {
            throw IllegalStateException("Accelerometer is not available")
        }
    }

    fun stopListening() {
        motionManager.stopAccelerometerUpdates()
    }
}
```

Классы у нас лежат в iOSMain и androidMain и написаны с использованием поддерживаемых SDK-платформ. Теперь нам необходимо объединить их в общее решение с некоторыми возможными изменениями.

Создадим собственный тип для акселерометра. Попробуем сначала обойтись без интерфейса, используя только `expect/actual` (листинг 3.29).

Листинг 3.29. Декларация общего типа

```
class Accelerometer {
  expect fun startListening(callback: (Float, Float, Float) -> Unit {}
  expect fun stopListening() {} // Modifier 'expect' is not applicable to
'member function'
}
```

Так сделать не получится, мы получим ошибку, потому что модификатором `expect` можно помечать либо декларации верхнего уровня, либо глобальные методы и свойства (рис. 3.6).

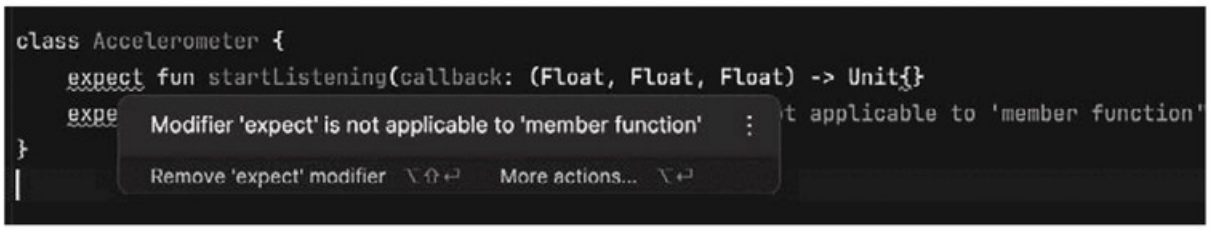


Рис. 3.6. Expect нужно использовать и на верхнем уровне

Можем попробовать реализовать так, как в листинге 3.30.

Листинг 3.30. Примерная декларация expect-класса для акселерометра

```
expect class Accelerometer {
  fun startListening(callback: (Float, Float, Float) -> Unit)
  fun stopListening()
}
```

Либо так, как в листинге 3.31, то есть с передачей блока обратного вызова в инициализатор класса.

Листинг 3.31. Вариант декларации expect-класса

```
expect class Accelerometer(listener: (x: Float, y: Float, z: Float) -> Unit) {
    fun startListening()
    fun stopListening()
}
```

Используем `expect`-модификатор для класса. Тогда все методы и свойства такого класса будут автоматически помечены `expect` и станут обязательными для реализации. То есть в данном случае не может быть реализации свойств или методов по умолчанию. Можем, конечно, попробовать что-то определить дополнительно в самих классах или файлах, но эти свойства и методы не будут цепляться к общей реализации. Например, для Android нам потребуется передать контекст приложения, что точно не нужно для iOS.

Поэтому будет правильнее использовать интерфейс для декларации и создания своего платформенного типа (листинг 3.32).

Листинг 3.32. Общий интерфейс в commonMain

```
interface Accelerometer {
  fun startListening(callback: (Float, Float, Float) -> Unit)
  fun stopListening()
}
```

`Expect/actual` можно использовать, чтобы маркировать глобальный метод для создания экземпляра полученного класса в iOSMain и androidMain (листинг 3.33).

Листинг 3.33. Expect функция-фабрика для создания инстанса типа

```
expect fun createAccelerometer(): Accelerometer
```

Создадим реализацию для Android (листинг 3.34).

Листинг 3.34. Фрагмент реализации для Android

```
private class AndroidAccelerometer : Accelerometer {
  override fun startListening(callback: (Float, Float, Float) -> Unit) {
    // Реализация из 3.27
  }
  override fun stopListening() {
    // Реализация из 3.27
  }
}

actual fun createAccelerometer(): Accelerometer = AndroidAccelerometer()
```

Чтобы не повторяться, здесь мы используем фрагменты из листинга 3.27.

Добавим для iOS аналогичную реализацию на основе листинга 3.28 (листинг 3.35).

Листинг 3.35. Фрагмент реализации класса для iOS

```
private class IOSAccelerometer : Accelerometer {

  override fun startListening(callback: (Float, Float, Float) -> Unit) {
    // Реализация из 3.28
  }

  override fun stopListening() {
    // Реализация из 3.28
  }
}

actual fun createAccelerometer(): Accelerometer = IOSAccelerometer()
```

Подключим наш код к Android так же, как мы это делали в предыдущем примере через объект-посредник для состояния (листинг 3.36).

Листинг 3.36. Подключение к посреднику на Android

```
class AccelerometerVM() {
  val text: MutableStateFlow<String> = MutableStateFlow("0:0:0")

  val accelerometer = createAccelerometer { x, y, z ->
    text.tryEmit("Accelerometer values: x=$x, y=$y, z=$z")
  }

  fun setup(context: Context) {
    accelerometer.setupContext(context)
    accelerometer.startListening()
  }
}

@Composable
fun AccelerometerScreen(vm: AccelerometerVM) {
  val text = vm.text.collectAsState()
  Text(text.value)
}
```

Контекст передадим из метода `onViewCreated` `Activity` (листинг 3.37).

Листинг 3.37. Передача контекста и модели состояния из Activity

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)

  val vm = AccelerometerVM()
  vm.setup(this)
  setContent {
    MyApplicationTheme {
      Surface(
        modifier = Modifier.fillMaxSize(),
        color = Color.White
      ) {
        AccelerometerScreen(vm)
      }
    }
  }
}
```

Проверим результат (рис. 3.7).

Альтернативно мы можем вынести логику работы в нативные приложения, реализовав интерфейс в классах Swift и Kotlin (листинг 3.38).

Листинг 3.38. Альтернативное решение на стороне iOS

```
class iOSAccelerometer: Accelerometer {
  func startListening(callback: @escaping (Float, Float, Float) -> Void) {}

  func stopListening() {}
}
```

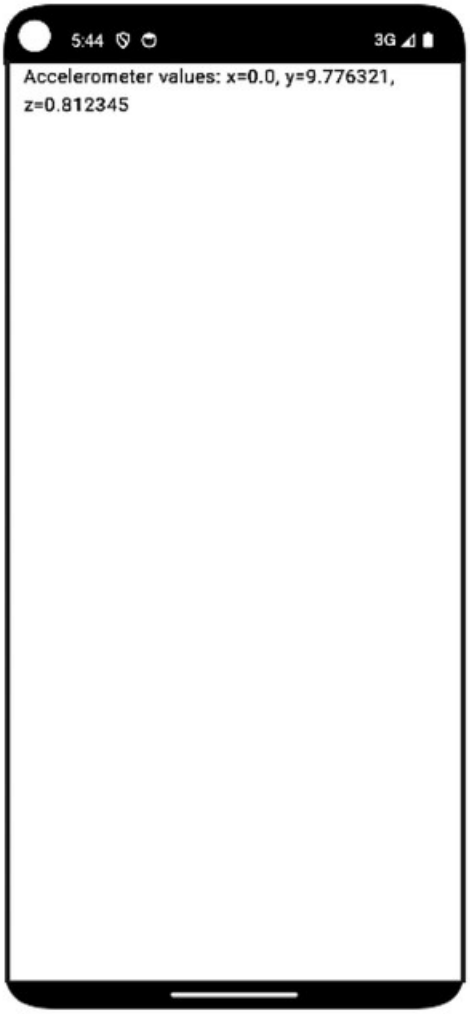


Рис. 3.7. Результат работы

В этом случае мы используем общий интерфейс из shared-модуля как базовый тип для работы с нативными реализациями.

Подключить такую логику можно либо через механизм DI (рассмотрим в главе 7), либо через метод компонента shared-логики, используя тот же общий интерфейс (листинг 3.39).

Листинг 3.39. Обертка для подключения компонента

```
class AccelerometerOwner {

  private var accelerometer: Accelerometer? = null

  fun setupAccelerometer(accelerometer: Accelerometer) {
    this.accelerometer = accelerometer
  }
}


accelerometerOwner.setupAccelerometer(iOSAccelerometer())
```

В целом для таких задач это менее гибкое решение.

Подробнее примеры сложных задач (например, как писать сетевой клиент) мы рассмотрим в следующих главах. В этой главе остановимся на простых примерах `expect/actual`.

Промежуточные выводы

Использование `expect/actual` в логике shared-модуля подойдет для переопределения свойств и функций глобального уровня. В остальных случаях будет удобнее использовать стандартный механизм с общим интерфейсом или комбинацию этих способов.

3.2.3. Платформо-специфичное поведение. Чтение файла

Предположим, мы пишем приложение, в котором необходимо считать содержимое файла и отобразить его на экране. Это может быть как некоторый встроенный файл с контентом, так и выбираемый пользователем на устройстве. В первом случае местоположение файла определяется ресурсами: для iOS это Bundle приложения, для android — каталог res. То есть пути к ресурсу у нас платформо-специфичные. Во втором случае файл подается со стороны нативного приложения.

По-разному определяется и механизм считывания содержимого файла в iOS SDK и Android SDK.

Для решения этой задачи выполним следующие действия:

1. Сначала объявим в commonMain класс `FileReader` с методом, в котором реализуем считывание. Помечаем класс как `expect` для дальнейшей реализации в нативных компонентах shared-модуля (листинг 3.40).

Листинг 3.40. Expect-декларация класса в commonMain

```
expect class FileReader constructor(){
  fun readFile(path: String): String
}
```

2. Напишем реализацию для Android в androidMain. Для работы с файлами в Kotlin JVM используется библиотека java.io.File (листинг 3.41).

Листинг 3.41. Реализация в androidMain

```
package ru.azharkova.filereader

import java.io.File

// Android модуль
actual class FileReader {
  actual fun readFile(path: String): String {
```

```
        return File(path).readText()
      }
    }
```

Получится стандартная реализация для JVM-приложений.

3. Теперь напишем реализацию для iOS (листинг 3.42). В iOS SDK не используется такая сущность, как File, чтобы получить доступ к реальному файлу. Мы используем строки NSString и через них считываем контент по заданному пути.

Листинг 3.42. Реализация FileReader для iOS

```
import kotlinx.cinterop.ExperimentalForeignApi
import platform.Foundation.NSString
import platform.Foundation.NSUTF8StringEncoding
import platform.Foundation.stringWithContentsOfFile

// iOS модуль
actual class FileReader {

  @OptIn(ExperimentalForeignApi::class)
  actual fun readFile(path: String): String {
    return NSString.stringWithContentsOfFile(path, encoding =
NSUTF8StringEncoding, error = null) as String
  }
}
```

Протестируем работоспособность решения на стороне iOS как на наименее родном и стабильном таргете.

Добавим в ViewModel нашего простого SwiftUI-приложения (листинг 3.43).

Листинг 3.43. Подключение к ViewModel на стороне iOS

```
import shared
import Foundation

class ContentVM : ObservableObject {
    @Published var text: String = ""

    func readFile() {
        let fileReader = FileReader()
        if let path = Bundle.main.path(forResource: "file", ofType: "txt") {
            let content = fileReader.readFile(path: path)
            print(content)
            text = content
        } else {
            print("File not found")
        }
    }
}
```

Теперь подключим чтение файла к ContentView, как в листинге 3.44.

Листинг 3.44. Вывод в SwiftUI View

```
struct ContentView: View {
    @ObservedObject var vm = ContentVM()
           let greet = Greeting().greet()

           var body: some View {
        Text(vm.text).onAppear {
            vm.readFile()
        }
           }
}
```

В качестве тестового файла будем использовать текстовый файл `file.txt`, который просто положим в наш проект и добавим в Bundle resources на вкладке Build phases. Текст на ваш выбор, для примера используем «Hello, Kotlin Multiplatform!».

Проверяем результат при запуске (рис. 3.8).



Рис. 3.8. Вывод результата

Аналогичные действия потребуется проделать, чтобы считать встроенный файл конфигурации и использовать его в работе нашего приложения.

3.2.4. Связь типов

Иногда возникает необходимость связывать разные типы Kotlin и Swift либо компоненты библиотеки под единой точкой вызова. Например, такое может произойти, если вам необходимо использовать механизм слабых ссылок для управления памятью. И в Kotlin, и в Swift этот механизм есть, называется одинаково, но работает по-разному в силу разницы между Android SDK и iOS SDK.

Создадим для начала общую декларацию `expect` для класса (листинг 3.45).

Листинг 3.45. Общая декларация для слабых ссылок

```
expect class WeakReference<T : Any>(referred: T) {
  fun clear()
  fun get(): T?
}
```

На iOS используем `actual`, но с указанием `typeAlias` (листинг 3.46).

Листинг 3.46. Actual-реализация для iOS

```
actual typealias WeakReference<T> = kotlin.native.ref.WeakReference<T>
```

Таким образом мы создаем привязку существующего типа к ссылке. В этом случае используем реализацию слабой ссылки из Kotlin Native.

Аналогично поступим с Android, создав связь с соответствующим типом (листинг 3.47).

Листинг 3.47. Actual-реализация для Android

```
actual typealias WeakReference<T> = java.lang.ref.WeakReference<T>
```

Здесь мы используем слабую ссылку из Kotlin JVM.

Теперь утечек в нашем коде точно не будет.

Еще один пример задачи такого рода. Нам необходимо использовать в shared-модуле интерфейс, который имеет значение для Android, но не для iOS, и из-за некоторых особенностей решения мы можем его обозначить только в commonMain, но не в androidMain. Например, Parcelable и Parcelize для data-классов.

В commonMain обозначим новую аннотацию и тип для привязки (листинг 3.48).

Листинг 3.48. Интерфейс и аннотация для общей декларации Parcelable

```
annotation class CommonParcelize

expect interface CommonParcelable
```

Для iOS создадим пустую реализацию без функционала в качестве заглушки (листинг 3.49).

Листинг 3.49. Actual-реализация заглушки для iOS

```
actual interface CommonParcelable
```

Для Android присвоим новому типу существующий интерфейс Parcelable (листинг 3.50).

Листинг 3.50. Actual-реализация для Android

```
actual typealias CommonParcelable = android.os.Parcelable
```

Теперь мы сможем использовать его с нашими data-классами из shared-модуля commonMain-части (листинг 3.51).

Листинг 3.51. Применение в commonMain

```
data class NewsItem(
        val author: String?,
        val title: String?,
        val description: String?,
        val url: String?,
        val urlToImage: String?,
        val publishedAt: String?,
        val content: String?
) : CommonParcelable
```

Мы рассмотрели основные примеры работы с нативным и общим кодом, а также способы правильно использовать `expect/actual` и интерфейсы для общего модуля. Конечно, это далеко не все возможные случаи и подходы. Например, некоторые команды разработки активно заняты подключением к общему модулю нативных низкоуровневых библиотек для работы с картами, звуком, видео, графикой для создания кросс-платформенных решений. Это могут быть работы в рамках официального портирования собственной библиотеки и по созданию инструмента для внутреннего использования. Как интегрировать нативный код в библиотеку KMP, мы сейчас рассматривать не будем. Но позже познакомимся с результатом работы одной из таких команд — Compose Multiplatform — и попробуем использовать его на практике.

3.3. Материал со звездочкой. Как работает под капотом интероп

Для того чтобы Kotlin-код из shared-модуля можно было использовать с iOS, его предварительно нужно скомпилировать во фреймворк. Давайте посмотрим на то, как преобразуется код Kotlin в Objective-C/Swift. Это пример из нашего shared-модуля. Чтобы ознакомиться с версией Objective-C, надо кликнуть по конструкции из shared, вызванной в iOSApp, и выбрать в контекстном меню Jump to Definition (листинг 3.52).

Листинг 3.52. Фрагмент кода на Objective-C из заголовочного файла в shared-модуле

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Greeting")))
@interface SharedGreeting : SharedBase
- (instancetype)init __attribute__((swift_name("init()")))
                     __attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
                      message="use object initializers instead")));
- (NSString *)greet __attribute__((swift_name("greet()")));
@end

__attribute__((swift_name("Platform")))
@protocol SharedPlatform
@required
@property (readonly) NSString *name __attribute__((swift_name("name")));
@end
```

Это фрагмент такого файла для примера.

В этом файле содержатся описания всех типов (протоколов, классов) с атрибутами, методами и свойствами, причем как базовых Kotlin-типов, так и сущностей, созданных нами в shared-модуле, поэтому размер и содержимое файла будут определяться вашим кодом приложения. Обратите внимание, что префикс Shared используется для тех типов, которые необходимо привести к типам Kotlin (листинг 3.53).

Листинг 3.53. Фрагмент кода на Objective-C из заголовочного файла в shared-модуле

```
__attribute__((swift_name("KotlinByte")))
@interface SharedByte : SharedNumber
- (instancetype)initWithChar:(char)value;
+ (instancetype)numberWithChar:(char)value;
@end

__attribute__((swift_name("KotlinUByte")))
@interface SharedUByte : SharedNumber
- (instancetype)initWithUnsignedChar:(unsigned char)value;
+ (instancetype)numberWithUnsignedChar:(unsigned char)value;
@end

__attribute__((swift_name("KotlinShort")))
@interface SharedShort : SharedNumber
- (instancetype)initWithShort:(short)value;
+ (instancetype)numberWithShort:(short)value;
@end
```

Например, в данном фрагменте создается описание некоторых числовых типов и `byte`. Параметр `swift_name` используется для задания наименования типа в Swift-версии этого кода.

Для ознакомления со Swift-версией нужно перейти к определению через контекстное меню в импорте, кликнув по shared (листинг 3.54).

Листинг 3.54. Конвертация кода в Swift

```
public class Greeting: KotlinBase {

  public init()

  open func greet() -> String
}

public protocol Platform {

  var name: String { get }
}

public class IOSPlatform: KotlinBase, Platform {

  public init()

  open var name: String { get }
}

public class Platform_iosKt: KotlinBase {

  open class func getPlatform() -> any Platform
}
```

Библиотеки на Swift поддерживаются, только если их API экспортирован в Objective-C с помощью `@objc`. Напрямую чистые Swift-модули в Kotlin-коде не поддерживаются.

Мы можем определить, какие части Kotlin-кода не портировать на Objective-C/Swift. Для этого надо пометить их с помощью аннотации `@HiddenFromObjC` (листинг 3.55).

Листинг 3.55. Сокрытие от портирования с помощью @HiddenFromObjC

```
@HiddenFromObjC
fun myKotlinOnlyFunction(){
  println("Only Kotlin!")
}
```

Если мы попробуем вызвать такой скрытый метод со стороны iOS, то получим ошибку (листинг 3.56).

Листинг 3.56. Пример ошибки при вызове скрытого метода

```
func hiddenFromObjCExample() {
  //Uncommenting gives compilation error
  //myKotlinOnlyFunction()
}
```

Если мы хотим заместить Kotlin-декларации обертками на Swift, то надо использовать аннотацию `@ShouldRefineInSwift`. Она помечает функцию или свойство

как `swift_private` в сгенерированном Objective-C API. Такие декларации также получают _ префикс и становятся невидимыми со стороны Swift. Например, код на Kotlin (листинг 3.57)

Листинг 3.57. Влияние @ShouldRefineInSwift

```
interface Person {
  @ShouldRefineInSwift
  val namePair: Pair<String, String>
}
class RealPerson: Person {
  override val namePair = "First" to "Last"
}
```

преобразуется в код Swift следующего вида (листинг 3.58):

Листинг 3.58. Преобразование из Kotlin в Swift

```
extension Person {

  // Переопределение свойства
  var name: (firstName: String, lastName: String) {
    let namePair = __namePair
    return (namePair.first! as String, namePair.second! as String)
  }
}

func shouldRefineInSwiftExample() {
  let authorNames = RealPerson().name
  print("Author is: \(authorNames.firstName) \(authorNames.lastName)")
}
```

Сейчас у нас нет перехода к Swift из Kotlin Native напрямую. Этот функционал на текущий момент находится в стадии разработки, и мы рассмотрим его в конце главы. Пока нам доступно преобразование в Objective-C сначала через язык C, а потом из Objective-C в Swift (рис. 3.9).

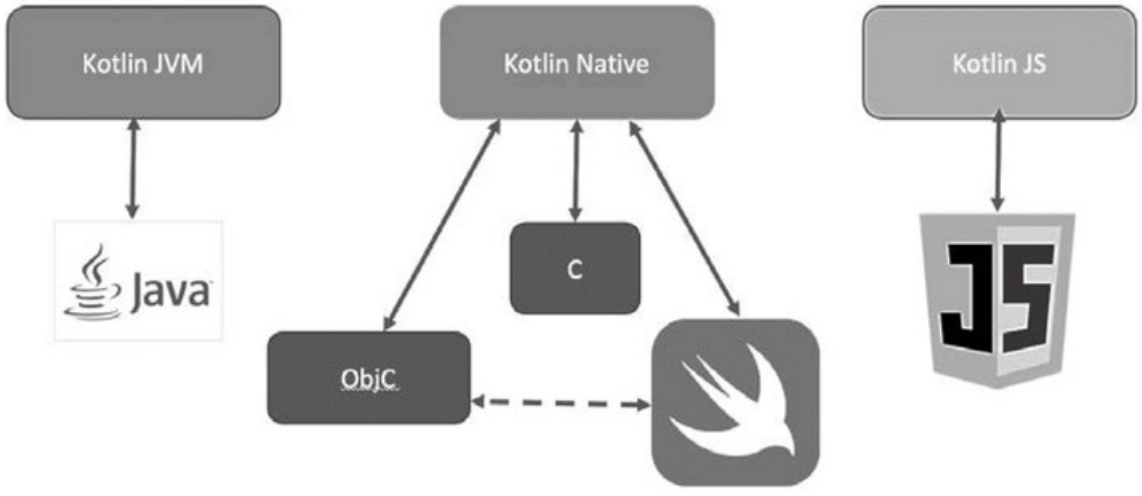


Рис. 3.9. Примерная схема преобразования Kotlin в Swift

Давайте рассмотрим этот процесс поэтапно (рис. 3.10).

1. Преобразование Kotlin-классов в код, поддерживаемый таргетами, происходит следующим образом. Сначала проходит фаза анализа и парсинга до абстрактного синтаксического древа с семантической информацией об узлах. Затем генерируется так называемое промежуточное представление IR, единое для всех платформ. Полученное абстрактное представление кода затем оптимизируется и компилируется в машинный код.

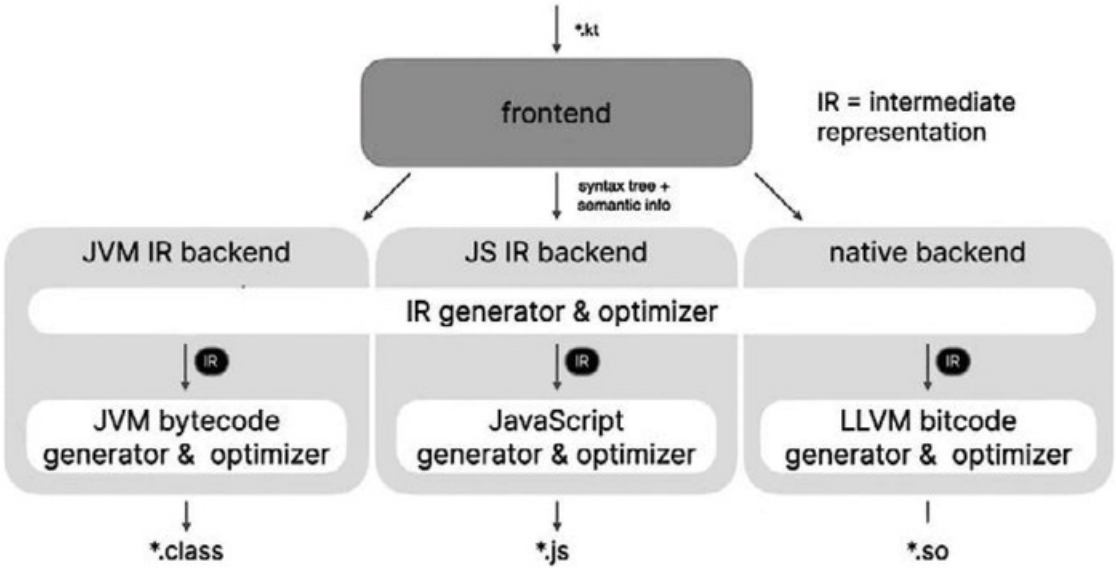


Рис. 3.10. Схема компиляции Kotlin-файлов для разных таргетов

2. Промежуточное представление IR поступает на так называемый IR Backend, определяемый таргетом. Здесь код пропускается через IR-генератор и оптимизацию. Далее с помощью соответствующей таргету виртуальной машины (в случае Kotlin Native LLVM) создается LLVM IR. LLVM IR — это низкоуровневое представление кода, которое может быть скомпилировано в машинный код для различных платформ.

   На этом этапе Kotlin/Native также генерирует необходимые runtime-библиотеки, которые обеспечивают работу Kotlin-кода (например, управление памятью, поддержку корутин и т. д.).

3. LLVM компилирует LLVM IR в нативный код для целевой платформы (например, ARM для iOS или x86_64 для macOS). На этом этапе генерируются объектные файлы (`.o` и `.so`), которые содержат нативный машинный код.

Объектные файлы, метаданные и ресурсы упаковываются в файл `klib`. Этот файл содержит:

- скомпилированный нативный код;
- метаданные о классах, функциях и свойствах;
- ресурсы (например, изображения или файлы конфигурации).

Вы можете проверить содержимое пакета `klib` (рис. 3.11), найдя его в списке внешних библиотек Gradle по имени. Открыв иерархию каталогов, можно увидеть default — ir со сгенерированными промежуточными представлениями типов, файлов, деклараций и т. д.

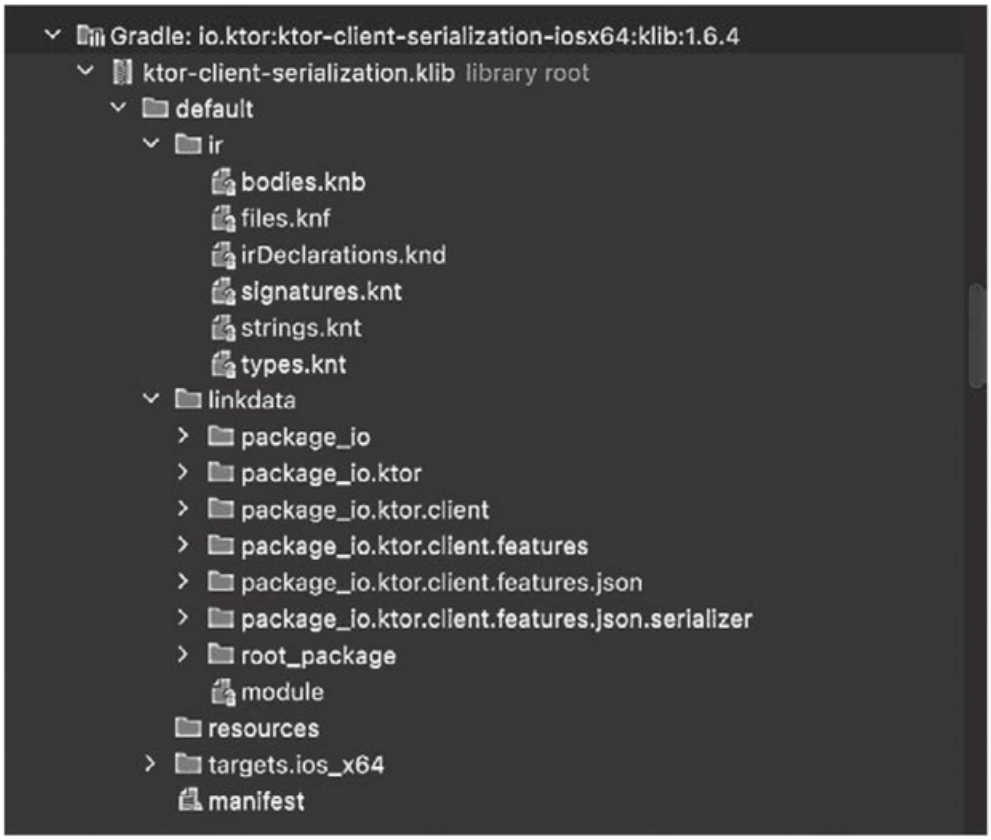


Рис. 3.11. Структура библиотеки klib

Если Kotlin Native использует C-библиотеки, процесс компиляции включает дополнительные этапы для обработки C-заголовочных файлов и генерации Kotlin-оберток (рис. 3.12).

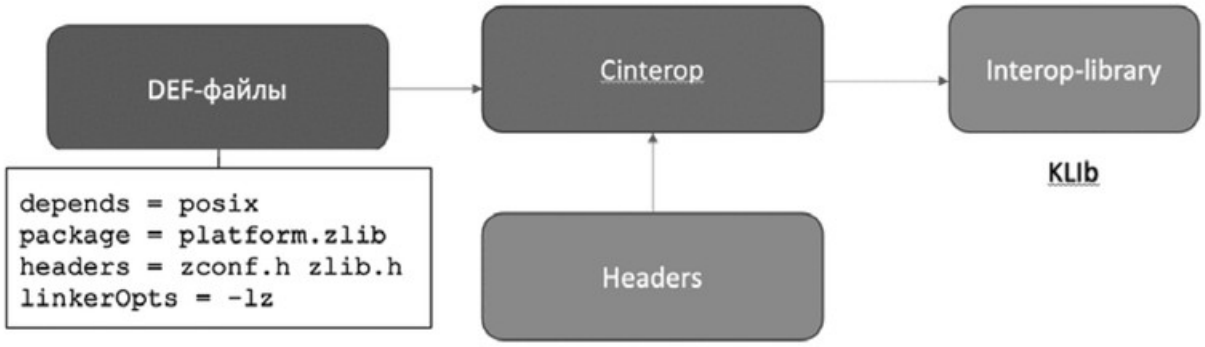


Рис. 3.12. Этапы для встраивания C-кода через cinterop

Это возможно в случае, если мы хотим встроить существующую библиотеку под iOS, особенно для C/Objective-C в `klib`. Например, что-то низкоуровневое или готовые решения по картам (например, как в Yandex-карте).

Далее создается `.def`-файл. `.def`-файл содержит информацию о том, какие C-заголовочные файлы (`*.h`) нужно обработать и как преобразовать C-функции в Kotlin-функции.

Следующим этапом идет обработка C-заголовочных файлов. Kotlin Native использует инструмент cinterop, который анализирует C-заголовочные файлы и генерирует Kotlin-обертки для C-функций, структур и других элементов.

Итоговые сгенерированные Kotlin-обертки компилируются вместе с Kotlin-кодом в промежуточное представление (IR), а затем в LLVM IR и нативный код. Результатом является `klib`, который содержит как Kotlin-код, так и обертки для C-библиотек.

В современной версии компилятора Kotlin K2 используется не только IR backend, но и специальный IR frontend, ускоряющий процесс компиляции и преобразования (рис. 3.13).

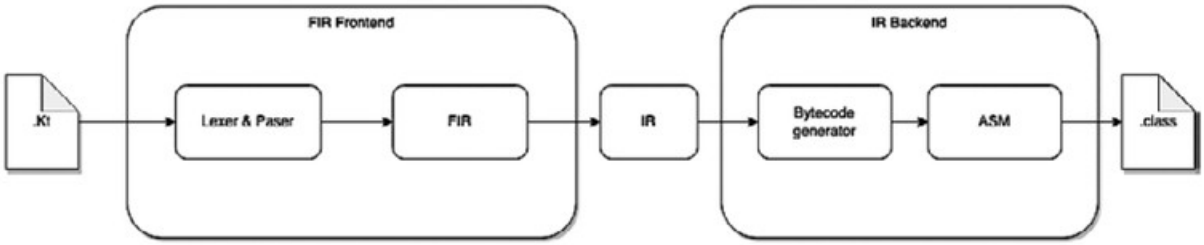


Рис. 3.13. Примерная схема компиляции K2

Стоит отметить, что LLVM генерирует код сугубо под определенную архитектуру iOS-устройства. На рис. 3.14 представлена схема работы LLVM-оптимизации с обработкой промежуточного представления LLVM IR под нужные таргеты.

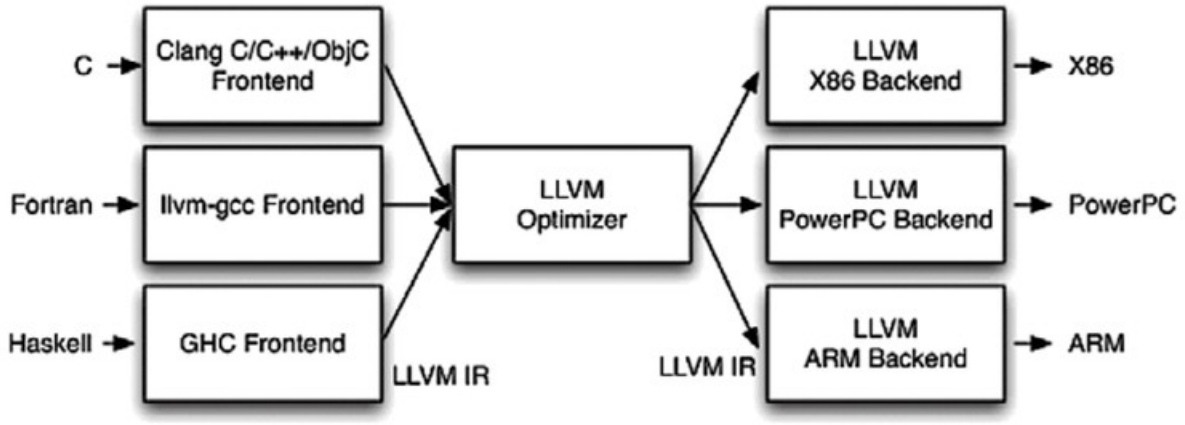


Рис. 3.14. Оптимизация промежуточного представления LLVM

Таким образом, мы получаем разные файлы `klib`. И чтобы корректно работать через единый таргет iOSMain с commonMain, используется механизм Commonizer (рис. 3.15).

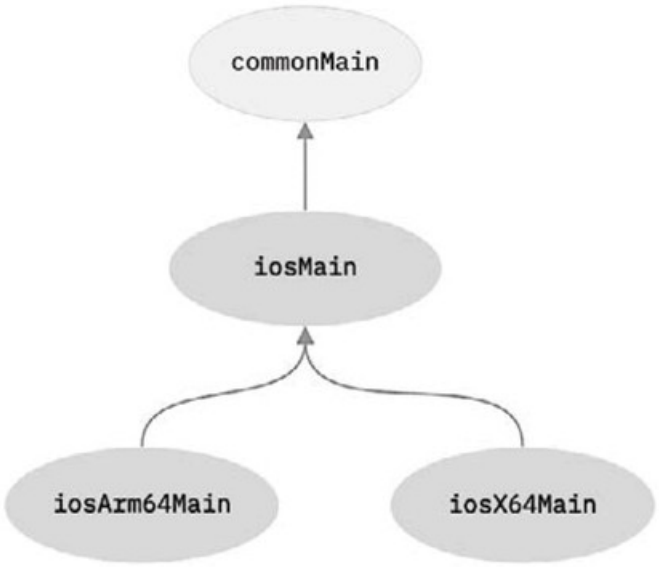


Рис. 3.15. Commonizer для iOS-таргетов

Он позволяет автоматически создавать общие модули для нескольких целевых платформ, что особенно полезно при разработке кросс-платформенных приложений. Давайте разберем, как работает Commonizer и как он взаимодействует с `klib` в Kotlin Native.

Commonizer изучает метаданные и API каждой платформо-специфичной библиотеки (klib). Он определяет, какие части кода могут быть обобщены (например, функции, классы, свойства), а какие являются платформо-специфичными. На основе анализа Commonizer создает общий модуль, который содержит только те API, которые доступны на всех целевых платформах. Этот общий модуль может быть использован в общем коде (common code) Kotlin Multiplatform. Для частей кода, которые не могут быть обобщены, Commonizer генерирует адаптеры. Эти адаптеры позволяют использовать платформо-специфичный код в общем модуле.

Мы получили `klib`. Но для iOS-проекта нам нужно получить framework. Для этого наш `klib` пройдет следующие этапы:

1. Kotlin Native компилирует `klib` в нативный код для целевой платформы (например, iosArm64 для устройств iOS или iosX64 для симулятора). На этом этапе генерируются объектные файлы (`.o`), которые содержат машинный код.
2. Объектные файлы объединяются в динамическую библиотеку (`.dylib`). Эта библиотека содержит скомпилированный Kotlin-код и все необходимые зависимости.

3. Kotlin Native автоматически генерирует заголовочные файлы (`.h`) для Objective-C и Swift. Эти файлы содержат объявления классов, методов и свойств, которые доступны в Kotlin-коде.
4. Kotlin Native создает структуру каталогов, которая соответствует стандарту Framework. В эту структуру добавляются:
   - динамическая библиотека (`.dylib`);
   - заголовочные файлы (`.h`);
   - ресурсы (если они есть).

Также генерируется файл `Info.plist`, который содержит метаданные о фреймворке (например, версию, идентификатор).

Сгенерированный фреймворк мы можем найти в каталоге build shared-модуля. Обратите внимание, что он собран под конкретную архитектуру. В данном случае мы запускали код на симуляторе arm64 (рис. 3.16).

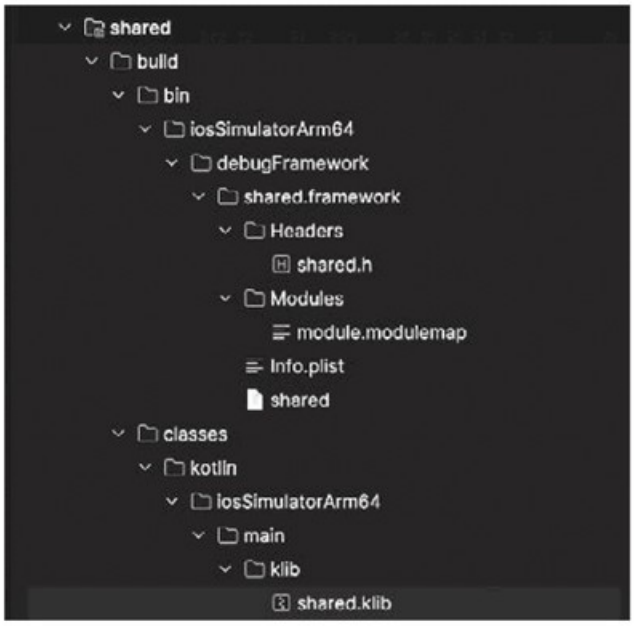


Рис. 3.16. Структура сгенерированного shared.framework

Давайте теперь посмотрим, что находится внутри сгенерированных фреймворк-файлов. Заголовочный файл содержит ту самую информацию, к которой мы переходили в Xcode из нативного iOS-приложения через Jump to Definition (листинг 3.59).

Листинг 3.59. Фрагмент кода заголовочного файла

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("FileReader")))
@interface SharedFileReader : SharedBase
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__
                                  ((objc_designated_initializer));
```

```
+ (instancetype)new __attribute__((availability(swift, unavailable,
                message="use object initializers instead")));
- (NSString *)readFilePath:(NSString *)path __attribute__
                           ((swift_name("readFile(path:)")));
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Greeting")))
@interface SharedGreeting : SharedBase
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__
                                   ((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
                message="use object initializers instead")));
- (NSString *)greet __attribute__((swift_name("greet()")));
@end
```

Это фрагмент содержимого, оригинал в случае приложения FileReader содержит около 185 строк.

Посмотрим теперь на `module.modulemap` нашего фреймворка shared-модуля (листинг 3.60).

Листинг 3.60. Файл module.modulemap

```
framework module "shared" {
  umbrella header "shared.h"

  export *
  module * { export * }

  use Foundation
}
```

Внутри мы видим подключение `shared.h` заголовочного файла через umbrella header. Этот механизм обеспечит доступность сгенерированного кода на Objective-C для Swift. Также автоматически включается экспорт всех модулей и подмодулей с зависимостью от Foundation-фреймворка.

Файл shared представляет собой бинарный файл нашего модуля. `Info.plist` содержит стандартный контент для файлов такого типа в iOS-проектах и библиотеках (листинг 3.61).

Листинг 3.61. Пример Info.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
                        PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>CFBundleExecutable</key>
    <string>shared</string>
    <key>CFBundleIdentifier</key>
    <string>ru.azharkova.filereader.shared</string>
```

```
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>shared</string>
    <key>CFBundlePackageType</key>
    <string>FMWK</string>
    <key>CFBundleShortVersionString</key>
    <string>1.0</string>
    <key>CFBundleSupportedPlatforms</key>
    <array>
      <string>iPhoneSimulator</string>
    </array>
    <key>CFBundleVersion</key>
    <string>1</string>
    <key>MinimumOSVersion</key>
    <string>14.0</string>
    <key>UIDeviceFamily</key>
    <array>
      <integer>1</integer>
      <integer>2</integer>
    </array>
  </dict>
</plist>
```

3.4. Из Kotlin в Swift и обратно

На KotlinfConf 2024 разработчики SDK сделали анонс экспериментального прямого экспорта Kotlin-кода в Swift. Подробнее можно посмотреть в проекте: https://github.com/Kotlin/swift-export-sample.

Этот функционал сейчас находится в экспериментальной стадии разработки, и разработчики явно предупреждают, что, во-первых, код будет меняться в следующих версиях, а во-вторых, он нестабилен, находится на ранней стадии разработки и не готов к работе в продакшене. Поэтому использовать его можно только на свой страх и риск.

Согласно тикету https://youtrack.jetbrains.com/issue/KT-64572, основной целью разработчиков является выпуск версии, которая имеет набор функций, аналогичный Objective-C export.

Вот некоторые из заявленных на данный момент целей:

- ограничение экспорта интерфейсов в качестве протоколов (сейчас внедрение Swift-типов нецелесообразно);
- поддержка функциональных типов в параметрах функций;
- обеспечение базовой поддержки универсальных типов с помощью их стирания.

За февраль 2025 года добавили базовую поддержку классов: `open`, `abstract`, `sealed`, `enum` и `inner`. Также появилась поддержка опционалов и нуллабильности, коллекций `List`, `Map` и `Set` и объявлений `@Deprecated` и `@Throws`.

Сейчас экспериментальный код поддерживается, начиная с Kotlin-версии 2.1.0 и выше. Для того чтобы попробовать прямой экспорт в своем проекте, нужно для начала настроить конфигурацию shared-модуля в блоке `kotlin {}` `build.gradle.kts`. Давайте посмотрим на листинг 3.62.

Листинг 3.62. Подключение экспорта в Swift для отдельных модулей

```
kotlin {
  iosX64()
  iosArm64()
  iosSimulatorArm64()

  @OptIn(ExperimentalSwiftExportDsl::class)
  swiftExport {
    // Root module name
    moduleName = "Shared"
    // Collapse rule
    flattenPackage = "com.github.jetbrains.swiftexport"
    // Export external modules
    export(projects.moduleA) {
      // Exported module name
      moduleName = "ModuleA"
      // Collapse exported dependency rule
      flattenPackage = "com.github.jetbrains.modulea"
    }
    export(projects.moduleB) {
      // Exported module name
      moduleName = "ModuleB"
      // Collapse exported dependency rule
      flattenPackage = "com.github.jetbrains.moduleb"
    }
  }
  sourceSets.commonMain.dependencies {
    api(projects.moduleA)
    api(projects.moduleB)
  }
}
```

Здесь `moduleA` и `moduleB` — экспортируемые из Kotlin в Swift модули. Для каждого из них мы используем инструкцию `export`, с указанием имени модуля при экспорте и пакета. Также добавляем эти модули в качестве зависимостей при компиляции API.

Для включения поддержки в проекте нужно в `gradle.properties` добавить следующие инструкции (листинг 3.63):

Листинг 3.63. Настройки для экспорта в gradle.properties

```
#MPP
kotlin.mpp.enableCInteropCommonization=true
kotlin.experimental.swift-export.enabled=true
kotlin.apple.xcodeCompatibility.nowarn=true
```

В нативном iOS-приложении тоже необходимо включить поддержку преобразования. Для этого в Run script добавим следующий код (листинг 3.64):

Листинг 3.64. Скрипт для подключения и сборки модуля Kotlin с поддержкой экспорта в Swift

```
if [ "YES" = "$OVERRIDE_KOTLIN_BUILD_IDE_SUPPORTED" ]; then
  echo "Skipping Gradle build task invocation due to OVERRIDE_KOTLIN_BUILD_IDE_
SUPPORTED environment variable set to \"YES\""
  exit 0
fi
cd "$SRCROOT/.."
./gradlew :shared:embedSwiftExportForXcode --configuration-cache
```

Обратите внимание, что мы заменили им предыдущую версию скрипта (листинг 3.65).

Листинг 3.65. Исходный скрипт для сборки shared-модуля

```
cd "$SRCROOT/.."
./gradlew :shared:embedAndSignAppleFrameworkForXcode
```

> **ВАЖНО!** Перед началом адаптации ознакомьтесь с актуальной документацией по поддержке этого функционала и с описанием особенностей работы с ним. Из-за того что в процессе доработки может меняться код подключения, некоторые инструкции могут устареть (причем внезапно) и перестать работать.

Посмотрим, что у нас поменяется после включения прямого экспорта. Запустим проект iOS, выберем код и нажмем в контекстном меню Jump to Definition.

До включения открывается код на Objective-C (ниже фрагмент, в оригинале 175 строк, листинг 3.66).

Листинг 3.66. Фрагмент исходного заголовочного файла на Objective-C

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Greeting")))
@interface SharedGreeting : SharedBase
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__
                                  ((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
                     message="use object initializers instead")));
- (NSString *)greet __attribute__((swift_name("greet()")));
@end

__attribute__((swift_name("Platform")))
```

```
@protocol SharedPlatform
@required
@property (readonly) NSString *name __attribute__((swift_name("name")));
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("IOSPlatform")))
@interface SharedIOSPlatform : SharedBase <SharedPlatform>
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__
                                  ((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
                      message="use object initializers instead")));
@property (readonly) NSString *name __attribute__((swift_name("name")));
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Platform_iosKt")))
@interface SharedPlatform_iosKt : SharedBase
+ (id<SharedPlatform>)getPlatform __attribute__((swift_name("getPlatform()")));
@end

#pragma pop_macro("_Nullable_result")
#pragma clang diagnostic pop
NS_ASSUME_NONNULL_END
```

После включения экспорта мы получим примерно такой код на Swift (листинг 3.67).

Листинг 3.67. Заголовочный файл на Swift

```
@_exported import ExportedKotlinPackages
import KotlinRuntime
@_implementationOnly import SharedBridge_shared

public typealias Greeting = ExportedKotlinPackages.dev.carrion.kmpswiftexport.
                            Greeting

public func getPlatform() -> Swift.Never {
  ExportedKotlinPackages.dev.carrion.kmpswiftexport.getPlatform()
}

extension ExportedKotlinPackages.dev.carrion.kmpswiftexport {

  public final class Greeting: KotlinRuntime.KotlinBase {
    public override init() {
      let __kt = dev_carrion_kmpswiftexport_Greeting_init_allocate()
      super.init(__externalRCRef: __kt)
      dev_carrion_kmpswiftexport_Greeting_init_initialize__TypesOfArguments__
                                  Swift_UInt__(__kt)
    }

    public override init(
      __externalRCRef: Swift.UInt
```

```
    ) {
      super.init(__externalRCRef: __externalRCRef)
    }

    public func greet() -> Swift.String {
      return dev_carrion_kmpswiftexport_Greeting_greet(self.__externalRCRef())
    }
  }

  public static func getPlatform() -> Swift.Never {
    fatalError()
  }
}
```

И это весь код: 175 строк кода превращаются в 28 строк существенно более читабельного кода. При этом большая часть информации о содержимом инкапсулирована и вынесена из заголовочного файла.

О планах и целях по поддержке Swift Export вы можете подробно прочитать здесь: https://youtrack.jetbrains.com/issue/KT-64572.

ПОДВЕДЕМ ИТОГИ

В этой главе мы подробно рассмотрели:

- ✓ как работать с `expect/actual` на различных примерах;
- ✓ какие ограничения есть при работе с кодом Kotlin для iOS и как их обойти;
- ✓ что происходит под капотом преобразования Kotlin в Swift/Objective-C.

Теперь мы готовы к реализации и подключению серьезной логики в наш каркас. Но прежде чем приступить к ней, мы погрузимся еще в одну важную тему, а именно многопоточность и управление памятью в KMP. Если вы считаете, что готовы идти дальше, переходите к главе 4.

ГЛАВА 4

Многопоточность в приложениях Kotlin Multiplatform

В этой главе рассмотрим:

- нюансы и различия в работе многопоточности на Android и iOS;
- реализацию общей работы с многопоточностью;
- способы подключения общих решений для работы с многопоточностью к Kotlin Native и iOS Swift;
- работу многопоточности под капотом.

Важный аспект реализации современного мобильного приложения — многопоточность и асинхронность. Многопоточность — это возможность программы выполнять несколько задач одновременно в рамках одного процесса. Данный функционал позволяет эффективно использовать вычислительные ресурсы компьютера или мобильного устройства и повышать производительность приложения. В контексте мобильной разработки многопоточность может быть особенно полезна, например, при загрузке данных из сети, включая параллельную обработку запросов, работу с графикой, сохранение в локальное хранилище, или при выполнении других трудоемких задач. Все тяжелые и затратные по ресурсам операции переносятся в фоновый поток, а результат выполнения при необходимости передается в основной (Main) поток приложения.

В случае приложений KMP мы чаще всего будем использовать многопоточность для осуществления сетевых запросов.

В предыдущей главе мы рассмотрели особенности и различия работы таргетов. Нативно различные таргеты приложения KMP используют разные механизмы реализации многопоточности и асинхронности. Для Android и JVM это Java-потоки, корутины Kotlin. Для iOS — это механизм Grand Central Dispatch, потоки и примитивы NSThread, фреймворк Operations, структурированная многопоточность `async/await`. Также стоит упомянуть `Combine` — реактивный API Swift (нативная замена RxSwift). Но как организовать общую работу для такого разного инструментария?

Kotlin Multiplatform позволяет настроить инструменты многопоточности в общем shared-модуле. Так как общий модуль работает с Kotlin, самым простым и удобным способом для организации асинхронной работы в приложении являются Kotlin `Coroutines`.

Начнем с подключения корутин в проект KMP (листинг 4.1).

Листинг 4.1. Подключение Kotlin Coroutines

```
commonMain {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:*.*.*")
    }
}
```

Добавим в блок для зависимостей commonMain ссылку на библиотеку `Kotlinx Coroutines Core`.

Для Android-таргета работа с корутинами в shared-модуле ничем не отличается от реализации в обычном нативном приложении.

4.1. Coroutines для iOS-разработчиков

Далеко не все разработчики iOS имеют опыт работы с `Coroutines` или знают, что это такое, поэтому стоит немного рассказать про этот механизм. На самом деле даже не все Android-разработчики знают, что это такое и как правильно с этим работать, так как могут использовать Rx и/или просто не вдаваться в детали.

Корутины — это одна из концепций многопоточности, которая существенно отличается от классического подхода с использованием потоков. Вместо того чтобы создавать отдельный поток для каждой задачи, корутины позволяют писать асинхронный код, который выглядит как синхронный. Это делает код более читаемым и упрощает управление состоянием. Важно понимать, что корутины не заменяют потоки, но представляют собой более высокий уровень абстракции для работы с асинхронностью и многопоточностью. Они могут быть особенно полезны в ситуациях, когда у вас есть множество небольших задач, которые нужно выполнять параллельно.

Проще говоря, корутина — это легковесный блок кода, который можно приостановить, не блокируя поток.

У корутины есть контекст выполнения (`CoroutineContext`) и область видимости, или скоуп (`CoroutineScope`). Контекст представляет собой описание элементов, которые входят в окружение корутины и управляют ее поведением, скоуп — это более высокоуровневая абстракция над контекстом. Основной задачей контекста является задание условий, как и где выполняется корутина, в то время как скоуп управляет жизненным циклом всех запущенных в нем корутин и их иерархией.

Жизненный цикл корутины контролируется с помощью `Job` (запуск, отмена, ожидание). Это объект, который представляет собой задачу, выполняемую корутиной. `Job` может быть в состоянии:

- active — корутина выполняется;
- completed — корутина завершила выполнение;
- cancelled — корутина была отменена.

`Job` также может быть частью иерархии, где отмена родительского `Job` приведет к отмене всех дочерних `Job`.

Для определения потока, в котором будет выполняться корутина, используется диспетчер корутин `CoroutineDispatcher`. Kotlin предоставляет несколько встроенных диспетчеров:

- `Dispatchers.Main` — для работы с UI (основным потоком в Android);
- `Dispatchers.IO` — для выполнения операций ввода-вывода (например, сетевые запросы, работа с файлами);
- `Dispatchers.Default` — для CPU-интенсивных задач (например, сортировка, тяжелые вычисления);
- `Dispatchers.Unconfined` — корутина будет выполняться в текущем потоке до первой точки приостановки, а затем продолжится в потоке, который вызвал возобновление. Этот диспетчер использовать не рекомендовано (кроме тестов), чтобы не получить блокировку при передаче ресурсов между потоками.

Обычно используются два диспетчера корутин:

- `Dispatchers.Main` — для работы/взаимодействия с UI-потоком;
- `Dispatchers.Default и Dispatchers.IO` — для работы и запросов в фоновом потоке. Dispatchers.Default используется для логики фоновой работы по умолчанию.

Информацию о `Job` и диспетчере корутин можно также получить из контекста.

Если проводить аналогию с iOS, то это похоже на выполнение блока кода в `DispatchQueue`, имеющей определенный QoS и привязку к определенному потоку `NSThread` либо `Operation` в `OperationQueue`, где `GlobalScope` аналогичен `DispatchQueue.global()`, а `MainScope` — `DispatchQueue.main` (листинг 4.2).

Листинг 4.2. Вызов асинхронного кода на Android и iOS

```
//Вызов асинхронного кода на Android
fun loadNews() {
  GlobalScope.launch {
    service.makeRequest()
    withContext(uiDispatcher) {
      //Вызов логики
    }
```

```
  }
}

//Вызов асинхронного метода на iOS
func loadNews() {
  DispatchQueue.global().async {
    service.makeRequest()
    DispatchQueue.main.async {
      //Вызов какой-то логики
    }
  }
}
```

Еще одной ключевой особенностью корутины является использование слова `suspend`. Этот модификатор не превращает метод в асинхронный сам по себе (это зависит от других деталей реализации), но маркирует, что его можно приостановить без блокировки потока. Также такой метод можно вызывать только в контексте корутины.

`GlobalScope` рекомендовано избегать, так как при его использовании возможны утечки памяти из-за трудностей с отслеживанием выполнения `Job` и зачистки ресурсов. Поэтому скоуп нужно создавать на выбранном диспетчере корутин (листинг 4.3).

Листинг 4.3. Создание скоупа и запуск корутины

```
val job = SupervisorJob() // Создание Job
val scope: CoroutineScope = CoroutineScope(ioDispatcher + job) //собственный
скоуп

//Android
fun loadNews() {
    scope.launch {
        service.makeRequest()
        withContext(uiDispatcher) {
            //Вызов логики
        }
    }
}
```

Чтобы ошибки в дочерних корутинах не сказывались на родительской, для запуска и работы с корутинами следует использовать скоуп, созданный с использованием `SupervisorJob`, который и позволяет создать такую иерархию задач.

Для Android есть поддержка готовых `lifecycleScopes`, привязанных к жизненному циклу того или иного объекта (`Activity`, `Fragment`). Такой функционал есть и для `ViewModel` (`viewModelScope`).

Для создания и запуска корутин используются специальные команды — конструкторы.

Если вам нужно запускать задачи, которые не требуют возврата результата, то используйте `launch`. Эта команда возвращает объект `Job`, с помощью которого можно управлять жизненным циклом задачи, в том числе и ее завершением (листинг 4.4).

Листинг 4.4. Запуск корутины с помощью launch

```
fun main() = runBlocking {
  val job = launch {
    delay(1000)
    println("Корутина завершена")
  }
  job.join() // Ожидаем завершения корутины
}
```

Если мы хотим запустить несколько задач одновременно, то следует использовать команду `async`. Результат запуска каждой корутины оборачивается в `Deferred<T>`, где `T` — тип возвращаемого значения. Дождаться результата всех задач можно с помощью `awaitAll()` (листинг 4.5).

Листинг 4.5. Запуск параллельных корутин с помощью async

```
fun main() = runBlocking {
  val deferred = async {
    delay(1000)
    "Результат"
  }
  println("Ожидаем результат...")
  println(deferred.await()) // Получаем результат
}
```

В примерах выше мы используем `runBlocking`. Это конструктор корутины, которая блокирует текущий поток до тех пор, пока не завершит выполнение собственной задачи. Будьте осторожны с его использованием. Он подойдет в основном для тестов.

С помощью команды `withContext` мы можем внутри одного скоупа вызвать блок с контекстом другого диспетчера и таким образом переключить контекст. Например, мы можем создать скоуп на диспетчере `Main` и обернуть вызов запроса сервиса в контекст фонового диспетчера (листинг 4.6).

Листинг 4.6. Переключения контекста корутин

```
fun loadNews() {
  CoroutineScope(uiDispatcher).launch {
    val result = withContext(ioDispatcher) {
      service.makeRequest()
    }
    // Обработка результата
  }
}
```

Механизм структурированной многопоточности `async/await` использует похожий подход с continuation, который также не блокирует поток, на котором вызывается.

Итак, на стороне shared-модуля `KotlinMultiplatform` мы оперируем `suspend`-методами и корутинами:

```
suspend fun getNewsList(): NewsList {
  return networkClient.request(Request(url = NEWS_LIST))
}
```

Что интересно, мы можем их вызывать напрямую из нашего iOS-приложения.

Благодаря поддержке interop Kotlin/Obj-C на стороне iOS-проекта, нативного с версии Kotlin 1.4, `suspend`-функции преобразуются в функции с `completion handler` (листинг 4.7).

Листинг 4.7. Конвертация Kotlin-кода в Objective-C

```
- (void)getNewsListWithCompletionHandler:(void (^)(SharedNewsList)
completionHandler __attribute__((swift_name
                                 ("getNewsList(completionHandler:)")));

//Swift
newsService.getNewsList(completionHandler: { response, error in
   if let data = response.articles {
    //...
   }

}
```

Далее в этом блоке мы можем вызвать вывод данных или выполнение какого-нибудь следующего метода — все стандартно и просто.

Однако не все любят простой синтаксис `completion handler`. А еще мы прекрасно знаем, что если им злоупотреблять, можно легко попасть в ситуацию callback hell[1] и потерять читабельность и чистоту кода.

Начиная с версии 1.5, Kotlin поддерживает автоматическую конвертацию `suspend` в `async/await`:

```
Task {
  let result = await newsService.getNewsList()
}
```

[1] Callback hell, или ад коллбэков, в программировании — это ситуация, когда код, использующий асинхронные операции через функции обратного вызова (callback), становится очень сложным для чтения и понимания из-за глубокой вложенности коллбэков (большого количества вложенных отступов). — *Примеч. ред.*

4.2. Совсем общая многопоточность. Настройка корутин для iOS и Kotlin/Native

Если мы реализуем в общем модуле только разные сервисы, то вполне можем обойтись вызовами их напрямую из нативных приложений. Однако если мы хотим реализовать приложение с общей архитектурой, общими вызовами и обработкой результатов запросов, нам потребуется настроить и общую работу с корутинами.

> **ПРИМЕЧАНИЕ** Чтобы корректно создавать скоуп из контекста корутины, потребуется задать диспетчер корутины.

Это платформозависимая логика, поэтому используем кастомизацию с помощью `expect/actual` (листинг 4.8).

Листинг 4.8. Декларация expect-диспетчеров корутин

```
expect val ioDispatcher: CoroutineContext


expect val uiDispatcher: CoroutineContext
```

`uiDispatcher` будет отвечать за работу в UI-потоке. `ioDispatcher` будем использовать для работы вне UI-потока.

AndroidMain работает на Kotlin JVM, а это значит, что мы можем использовать стандартные диспетчеры корутин. Для доступа к соответствующим потокам используем `CoroutineDispatchers Main` (UI-поток) и `IO` (стандартный для `Coroutine`). Наши `actual`-реализации для androidMain будут иметь такой вид (листинг 4.9):

Листинг 4.9. Реализация диспетчеров корутин в androidMain

```
actual val uiDispatcher: CoroutineContext

    get() = Dispatchers.Main


actual val ioDispatcher: CoroutineContext

    get() = Dispatchers.IO
```

Для фона мы можем также использовать `Dispatchers.Default`.

Давайте посмотрим, как реализуются сами диспетчеры корутин разного типа. Диспетчер `MainDispatcher` выбирается для платформы внутри `CoroutineDispatcher` с помощью фабрики диспетчеров `MainDispatcherLoader` (листинг 4.10).

Листинг 4.10. Устройство MainDispatcherLoader

```
internal object MainDispatcherLoader {

    private val FAST_SERVICE_LOADER_ENABLED = systemProp(FAST_SERVICE_LOADER_
                PROPERTY_NAME, true)

    @JvmField
    val dispatcher: MainCoroutineDispatcher = loadMainDispatcher()

    private fun loadMainDispatcher(): MainCoroutineDispatcher {
        return try {
            val factories = if (FAST_SERVICE_LOADER_ENABLED) {
                FastServiceLoader.loadMainDispatcherFactory()
            } else {
                // We are explicitly using the
                // `ServiceLoader.load(MyClass::class.java, MyClass::class.
                    java.classLoader).iterator()`
                // form of the ServiceLoader call to enable R8 optimization
                   when compiled on Android.
                ServiceLoader.load(
                        MainDispatcherFactory::class.java,
                        MainDispatcherFactory::class.java.classLoader
                ).iterator().asSequence().toList()
            }
            @Suppress("ConstantConditionIf")
            factories.maxBy { it.loadPriority }?.tryCreateDispatcher(factories)
                ?: createMissingDispatcher()
        } catch (e: Throwable) {
            // Service loader can throw an exception as well
            createMissingDispatcher(e)
        }
    }
}
```

Для IO-диспетчера используется готовый `DefaultIoScheduler` JVM-версии корутин (листинг 4.11).

Листинг 4.11. DefaultIoScheduler

```
// Dispatchers.IO
internal object DefaultIoScheduler : ExecutorCoroutineDispatcher(), Executor {

  private val default = UnlimitedIoScheduler.limitedParallelism(
      systemProp(
          IO_PARALLELISM_PROPERTY_NAME,
          64.coerceAtLeast(AVAILABLE_PROCESSORS)
      )
  )
  //...
}
```

Внутри этого механизма поддерживается использование максимально возможного количества процессоров. Оно зависит от архитектуры и доступных ресурсов.

Посмотрим, что мы можем применить для iOS. Для iOS-таргета в shared-модуле используются Kotlin/Native и свои диспетчеры корутин, реализованные с учетом механики многопоточности iOS SDK (листинг 4.12).

Листинг 4.12. Реализация диспетчеров для iOSMain

```
actual val uiDispatcher: CoroutineContext

    get() = Dispatchers.Main


actual val ioDispatcher: CoroutineContext

    get() = Dispatchers.IO
```

Да, сейчас это выглядит так же, как для Android. Попробуем посмотреть, что внутри. Для этого кликнем, например, по `Dispatchers.Main` и перейдем в контекстном меню к имплементации через Implementations. В зависимости от версии Kotlin и библиотеки для работы с корутинами мы увидим примерно такой код (листинг 4.13):

Листинг 4.13. Реализация MainDispatcher для Kotlin/Native

```
private class DarwinMainDispatcher(
        private val invokeImmediately: Boolean
) : MainCoroutineDispatcher(), Delay {

        override val immediate: MainCoroutineDispatcher =
                if (invokeImmediately) this else DarwinMainDispatcher(true)

        override fun isDispatchNeeded(context: CoroutineContext): Boolean =
!(invokeImmediately && isMainThread())

        override fun dispatch(context: CoroutineContext, block: Runnable) {
            autoreleasepool {
                dispatch_async(dispatch_get_main_queue()) {
                    block.run()
                }
            }
        }
}
```

`DarwinMainDispatcher` представляет собой реализацию абстрактного класса `MainCoroutineDispatcher`. Обратите внимание, что в его реализации метода dispatch используется инструментарий Grand Central Dispatch из iOS SDK: блок кода вызывается асинхронно в очереди `DispatchQueue.Main` (`dispatch_get_main_queue` — нотация ObjC), для управления памятью используется autoreleasepool.

В ранних версиях KMP и до версии Kotlin 1.7.20 `MainDispatcher` необходимо было создавать вручную (листинг 4.14).

Листинг 4.14. Ручное создание MainDispatcher

```
private object MainDispatcher: CoroutineDispatcher(){

    override fun dispatch(context: CoroutineContext, block: Runnable) {

        dispatch_async(dispatch_get_main_queue()) {

            try {

                block.run()

            }catch (err: Throwable) {

                throw err

            }

        }

    }

}
```

Теперь посмотрим, что у нас происходит с диспетчерами для фоновых задач. Сейчас для `Dispatchers.Default` используется собственный `CoroutineDispatcher`. Единственное отличие от `MainDispatcher` состоит в том, что вызов кода оборачивается в фоновую очередь `DispatchQueue.global()` или `dispatch_get_global_queue`, если смотреть нотацию ObjC (листинг 4.15).

Листинг 4.15. Устройство DarwinGlobalQueueDispatcher

```
private object DarwinGlobalQueueDispatcher : CoroutineDispatcher() {
    override fun dispatch(context: CoroutineContext, block: Runnable) {
        autoreleasepool {
            dispatch_async(
                dispatch_get_global_queue(
                    DISPATCH_QUEUE_PRIORITY_DEFAULT.convert(), 0u)) {
                block.run()
            }
        }
    }
}
```

До реализации новой модели управления памятью и версии Kotlin 1.7.20 портирование `dispatch_get_global_queue` на Kotlin/Native реализовано не было.

IO-диспетчер для Kotlin/Native выглядит несколько более сложным. Подробнее об этом см. в разделе со звездочкой (4.5).

4.3. Использование корутин и скоупов

Итак, мы создали диспетчеры. Теперь можем создать скоуп (область видимости корутины) и использовать его в работе. Наш скоуп должен также чем-то регулироваться и где-то вызываться. Если скоуп нужен внутри некоторого сервиса, декларируем его прямо там (листинг 4.16).

Листинг 4.16. Создание скоупа в сервисе

```
class SomeService {

  private val job = SupervisorJob()
  val scope: CoroutineScope = CoroutineScope(ioDispatcher + job)

}
```

В зависимости от выбранной нами архитектуры для компонента, через который мы будем взаимодействовать с UI, создадим базовый класс и в нем расположим наш скоуп. Например, в нашем случае это `ViewModel` (листинг 4.17).

Листинг 4.17. Создание диспетчера в базовой ViewModel

```
class BaseViewModel(val dispatcher = uiDispatcher) {

  val scope: CoroutineScope = CoroutineScope(dispatcher)

}
```

Для этого передадим диспетчер, на котором будет работать наш скоуп, через конструктор.

Теперь можем оборачивать вызовы `suspend`-методов в скоуп.

> **ВНИМАНИЕ!** В последних версиях KMP и Kotlin мы можем обойтись без `expect` и `actual` и объявить все нужные диспетчеры корутин для фонового и main потоков сразу в commonMain, чтобы безопасно вызывать их из нашей shared-логики (листинг 4.18).

Листинг 4.18. Реализация диспетчеров в commonMain

```
val uiDispatcher: CoroutineDispatcher = Dispatchers.Main

val ioDispatcher: CoroutineDispatcher = Dispatchers.IO

val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
```

Именно это мы и включим в наш код. Создадим каталог util, в который добавим файл `Dispatchers` с таким содержимым:



Рис. 4.1. Примерная иерархия каталогов для Dispatchers

Проверим работоспособность наших диспетчеров. Также создадим скоуп на одном из диспетчеров из commonMain (без `expect/actual`).

```
val scope: CoroutineScope = CoroutineScope(ioDispatcher)
```

Теперь добавим метод проверки, в котором попробуем вывести информацию о созданной корутине (листинг 4.19)

Листинг 4.19. Вывод информации о корутине

```
fun test() {
  scope.launch {
    print(this)
  }
}
```

и вызовем его из нашего iOS-приложения. Проверим вывод в консоль IDE:

```
StandaloneCoroutine{Active}@6f40160
```

Все действительно работает. Дело в том, что для подстановки нужных диспетчеров внутри библиотеки корутин по-прежнему используется механизм `expect/actual`. Просто теперь ответственность за реализацию расположена внутри решения.

4.4. Под капотом Dispatchers.IO

Если мы кликнем правой кнопкой мыши по `Dispatchers.IO` и перейдем в контекстном меню к имплементации через Implementations, то увидим `expect`-объявление нашего диспетчера (листинг 4.20).

Листинг 4.20. Код Dispatchers.IO

```
package kotlinx.coroutines

/**
 * The [CoroutineDispatcher] that is designed for offloading blocking IO tasks
   to a shared pool of threads.
 * Additional threads in this pool are created on demand.
 * Default IO pool size is `64`; on JVM it can be configured using JVM-specific
   mechanisms,
 * please refer to `Dispatchers.IO` documentation on JVM platform.
```

```
 *
 * ### Elasticity for limited parallelism
 *
 * `Dispatchers.IO` has a unique property of elasticity: its views
 * obtained with [CoroutineDispatcher.limitedParallelism] are
 * not restricted by the `Dispatchers.IO` parallelism. Conceptually, there is
 * a dispatcher backed by an unlimited pool of threads, and both
   `Dispatchers.IO`
 * and views of `Dispatchers.IO` are actually views of that dispatcher.
   In practice
 * this means that, despite not abiding by `Dispatchers.IO`'s parallelism
 * restrictions, its views share threads and resources with it.
 *
 * In the following example
 * ```
 * // 100 threads for MySQL connection
 * val myMysqlDbDispatcher = Dispatchers.IO.limitedParallelism(100)
 * // 60 threads for MongoDB connection
 * val myMongoDbDispatcher = Dispatchers.IO.limitedParallelism(60)
 * ```
 * the system may have up to `64 + 100 + 60` threads dedicated to blocking
   tasks during peak loads,
 * but during its steady state there is only a small number of threads shared
 * among `Dispatchers.IO`, `myMysqlDbDispatcher` and `myMongoDbDispatcher`
 */
@Suppress("EXTENSION_SHADOWED_BY_MEMBER")
public expect val Dispatchers.IO: CoroutineDispatcher
```

В комментариях к объявлению мы увидим пример по работе с заданием различного числа потоков исполнения.

Теперь посмотрим, что скрывает `Dispatchers.IO`, если его объявить в `iOSMain`. Переходим так же через Implementations, но выбираем реализацию именно нужного таргета.

Найдем `actual`-реализацию диспетчера с помощью `DefaultIoScheduler` (листинг 4.21).

Листинг 4.21. Реализация DefaultIoScheduler

```
Suppress("EXTENSION_SHADOWED_BY_MEMBER")
public actual val Dispatchers.IO: CoroutineDispatcher get() = IO

internal val IO: CoroutineDispatcher = DefaultIoScheduler
```

Для каждой из платформ-таргетов `DefaultIoScheduler` реализуется по-своему с учетом особенностей внутренней организации, в нашем случае Kotlin JVM и Kotlin Native.

Для создания фоновых диспетчеров корутин с поддержкой одного или нескольких потоков мы можем использовать специальные функции (листинг 4.22).

Листинг 4.22. Создание диспетчера корутин с фиксированным числом потоков

```
fun newFixedThreadPoolContext(nThreads: Int, name: String):
CloseableCoroutineDispatcher

//Однопоточная фоновая корутина
public fun newSingleThreadContext(name: String): CloseableCoroutineDispatcher =
           newFixedThreadPoolContext(1, name)
```

Вызов такого диспетчера аналогичен переключению контекста с помощью `withContext(…)`, поэтому обращаться к нему нужно внутри скоупа корутины (листинг 4.23).

Листинг 4.23. Запуск однопоточной корутины

```
scope.launch {
  newSingleThreadContext("single").invoke {
    // Код для выполнения
  }
}
```

В случае с Kotlin Multiplatform мы создаем скоуп для вызова корутин самостоятельно, но для вызова со стороны Android можем использовать компонент из библиотек `Jetpack ViewModel` со встроенным скоупом, который отслеживает, кстати, жизненный цикл самой `ViewModel`. В следующих главах мы рассмотрим работу с общей `ViewModel`, которая работает с корутинами.

4.5. Материал со звездочкой. Внутренняя кухня многопоточности в Kotlin/Native

Текущая реализация Kotlin/Native решает все возможные проблемы многопоточности за нас, значительно облегчая нам работу. Но чтобы понимать, как это работает, давайте посмотрим, что скрывает в себе `DefaultIoScheduler` в Kotlin/Native. Вы можете пропустить эту часть и перелистнуть страницы, перейдя к практическим примерам подключения Kotlin `Coroutines` в код существующего iOS-приложения.

Итак, под капотом `DefaultIoScheduler` скрывается диспетчер с фиксированным количеством потоков (листинг 4.24).

Листинг 4.24. DefaultIoScheduler под капотом

```
internal object DefaultIoScheduler : CoroutineDispatcher() {
  // 2048 is an arbitrary KMP-friendly constant
  private val unlimitedPool = newFixedThreadPoolContext(2048, "Dispatchers.IO")
  private val io = unlimitedPool.limitedParallelism(64) // Default JVM size

  @ExperimentalCoroutinesApi
  override fun limitedParallelism(parallelism: Int): CoroutineDispatcher {
```

```
    // See documentation to Dispatchers.IO for the rationale
    return unlimitedPool.limitedParallelism(parallelism)
  }

  override fun dispatch(context: CoroutineContext, block: Runnable) {
    io.dispatch(context, block)
  }
  //…
}
```

`newFixedThreadPoolContext` представляет собой `MultiWorkerDispatcher`. Помимо стандартного функционала корутин-диспетчеров (continuation), этот диспетчер использует следующие API:

- Worker для работы с фоновыми потоками;
- Atomics для корректной передачи ресурсов между потоками;
- Channels (примитив для связи между корутинами и синхронизации доступа к данным) для очередей воркеров и задач.

Каждый Worker представляет собой некую очередь работы, с помощью которой можно выполнять задачи в отдельных потоках; на каждый Worker создается свой отдельный поток. Для создания Worker достаточно вызвать:

```
Worker.start()
```

При самостоятельном использовании Worker в качестве средства реализации многопоточности без корутин и вне скоупа/диспетчера можно создать специальную функцию-обертку с передачей блока для обратного вызова (листинг 4.25).

Листинг 4.25. Реализация обертки для выполнения в фоне

```
internal fun background(block: () -> (Any?)) {
  // 1 Создаем future
  val future = worker.execute(TransferMode.SAFE, { block }) {
    // 2 Выполнение переданного кода
    it()
  }
  // 3 Создаем массив future для управления жизненным циклом
  collectFutures.add(future)
}

private val worker = Worker.start()
private val collectFutures = mutableListOf<Future<*>>()
```

`Future` создается при запуске функционального блока, который мы передаем в `worker.execute` или `worker.executeAfter` для выполнения в `producer`. Мы также можем отследить выполнение `future` с помощью сохранения в коллекцию ссылки на него (главное — не забыть потом очистить коллекцию от ссылок, чтобы не создавать утечки памяти). Чтобы наш блок выполнялся потокобезопасно, используем параметр TransferMode.SAFE.

В legacy-версиях модели управления памяти (до 1.7.20) требовалось использовать заморозку блока перед выполнением (листинг 4.26). Это позволяло сделать его неизменяемым.

Листинг 4.26. Заморозка блока

```
internal fun <T> T.share(): T {
    return this.freeze()
}
```

Получение результата работы `future` производится в специальном блоке `consume` (листинг 4.27).

Листинг 4.27. Функция-обертка с возвратом результата в основной поток

```
internal fun background(block: () -> (Any?), callback: (Any?) -> Unit) {
    val future = worker.execute(TransferMode.SAFE, { block }) {
        it()
    }
    future.consume {
        main {
            // Вызываем по завершении
            callback(it)
        }
    }
    collectFutures.add(future)
}
```

При самостоятельном использовании без корутин маршрутизацию в главный поток можно сделать с помощью обертки в `dispatch_get_main_queue` (листинг 4.28).

Листинг 4.28. Обертка для вызова блока кода в основном потоке

```
internal fun main(block:()->Unit) {

        dispatch_async(dispatch_get_main_queue()) {

            block()

        }}
```

В реализации `WorkerDispatcher` и `MultiWorkerDispatcher` используется обертка в continuation, куда передается результат работы (листинг 4.29).

Листинг 4.29. Фрагмент кода WorkerDispatcher

```
internal class WorkerDispatcher(name: String) : CloseableCoroutineDispatcher(),
Delay {
  private val worker = Worker.start(name = name)

  override fun dispatch(context: CoroutineContext, block: Runnable) {
    worker.executeAfter(0L) { block.run() }
  }
```

```
  override fun scheduleResumeAfterDelay(timeMillis: Long,
      continuation: CancellableContinuation<Unit>) {
    val handle = schedule(timeMillis, Runnable {
      with(continuation) { resumeUndispatched(Unit) }
    })
    continuation.disposeOnCancellation(handle)
  }
  //Продолжение реализации
}
```

Это пример кода из библиотеки `Kotlinx Coroutines`.

Для корректной работы с ресурсами в потоках можно использовать atomic API (например, AtomicInt, AtomicLong для числовых типов, AtomicNativePtr для указателей и т. п.). Для работы с объектами мы можем использовать AtomicReference.

Рассмотрим пример, где это может потребоваться. Представим, что ответ на сетевой запрос к нам приходит несколькими порциями и перед финальной конвертацией надо собрать все в один массив. Объявим его AtomicReference (листинг 4.30).

Листинг 4.30. Создание атомарной ссылки

```
// Обертка для атомарной ссылки
internal fun <T> T.atomic(): AtomicReference<T> {
  return AtomicReference(this)
}

private val chunks = ByteArray(0).atomic()

// Пример работы с атомарной ссылкой
private fun updateChunks(data: NSData) {
  //Использование временной переменной для пересоздания атомарной ссылки
  var newValue = ByteArray(0)
  newValue += chunks.value
  newValue += data.toByteArray()
  chunks.value = newValue
}
```

Итоговые данные мы можем отдать наружу с помощью `Channels` или `CompletableDeferred` (листинг 4.31). Этот функционал позволяет создать awaitable-поведение с помощью вызова метода `await` у `CompletableDeferred` и дождаться сигнала по вызову `complete`.

Листинг 4.31. Использование CompletableDeferred для awaitable-поведения

```
class DefferedResponseReader: NSObject(), NSURLSessionDataDelegateProtocol {
  private var chunks = ByteArray(0).atomic()
  private var rawResponse = CompletableDeferred<Response>()

  suspend fun awaitResponse(): Response {
    // Вызываем await, чтобы дождаться завершения CompletableDeferred
```

```
    return rawResponse.await()
  }

  override fun URLSession(
      session: NSURLSession,
      task: NSURLSessionTask,
      didCompleteWithError: NSError?
  ) {
    val response = task.response as NSHTTPURLResponse
    // Завершаем исполнение при ошибке
    completed(response.statusCode, didCompleteWithError as? Error)
  }
```

В качестве примера здесь использована реализация сетевого клиента iOS в iOSMain на Kotlin Native.

Еще одним возможным API для работы с многопоточностью является DetachedObjectGraph (листинг 4.32).

Листинг 4.32. Использование DetachedObjectGraph

```
//код Supranatural concurrency

    override fun <T, V> execute(jobInput: T, job: (T) -> V): Future<V> {
        //1 Определяем DeferredFuture для получения результата
        val deferred = DeferredFuture<V>()
        //2 Создаем DetachedObjectGraph с передачей параметров
            и указателя на него
        val detached = DetachedObjectGraph { Triple(jobInput, job,
                       deferred).toImmutable() }.asCPointer()

        //3 Передача графа для вызова асинхронно в очереди
        dispatch_async_f(dispatch_get_main_queue(), detached,
                         staticCFunction { it: COpaquePointer? ->
            initRuntimeIfNeeded()
            //4 Получение графа внутри очереди
            val attached = DetachedObjectGraph<Triple<T, (T) -> V,
                           DeferredFuture<V>>>(it).attach()
            //5 Получаем параметры из графа. В job передаем входные данные
            val result = attached.second(attached.first)
            //6 Возврат результата в Future
            attached.third.setValue(result)
        })
        return deferred
    }
```

Этот способ примечателен тем, что может передавать объекты между потоками без заморозки. Вот как это достигается:

1. Кладем входные параметры, ссылки на блоки и коллбэки, которые нужно вызвать и передать в другом потоке, в качестве общего содержимого внутрь DetachedObjectGraph.

2. Получаем указатель на откреплённый подграф с объектом с помощью asCPointer и внутри нужного потока вызываем staticCFunction, где в качестве параметра работаем с указателем.
3. Прикрепляем граф с помощью команды `attach`, извлекаем упакованные данные и преобразуем их нужным образом.
4. Возвращаем коллбэк вне текущего потока.

ВАЖНО! `Attach` откреплённого объекта можно вызвать один раз. Иначе можно получить исключение IllegalStateException: Illegal transfer state. Поэтому ссылка на граф должна обнуляться после окончания работы блока.

Также при упаковке объекта полезно указать TransferMode.SAFE. Но в API DetachedObjectGraph этот параметр используется по умолчанию.

Для удобства можно сделать обертку, которая позволит работать с изменяемыми элементами в разных потоках, соединив и Worker, и AtomicReference, и DetachedGraphObject (листинг 4.33).

Листинг 4.33. Обертка для работы с изменяемыми объектами в разных потоках

```
class SharedDetachedObject<T:Any>(producer: () -> T) {

    private val adog : AtomicReference<DetachedObjectGraph<Any>?>

    private val lock = Any()


    init {

        val detachedObjectGraph = DetachedObjectGraph { producer() as Any }

        adog = AtomicReference(detachedObjectGraph)

    }


    fun <R> access(block: (T) -> R): R = trySynchronized(lock){

        val holder = AtomicReference<Any?>(null)

        val producer = { grabAccess(holder, block) as Any }

        adog.value = DetachedObjectGraph(TransferMode.SAFE, producer)

        val result = holder.value!!

        holder.value = null

        result as R

    }
```

```
    private fun <R> grabAccess(holder: AtomicReference<Any?>, block: (T) ->
                R):T{

        val attach = adog.value!!.attach()

        val t = attach as T

        holder.value = block(t)

        return t

    }

}
```

ПРИМЕЧАНИЕ Это решение вам может пригодиться при организации собственного низкоуровневого решения для работы с многопоточностью.

В настоящий момент все API, связанные с прямым использованием Worker и WorkerDispatcher, уже помечены как Obsolete:

```
@file:OptIn(ObsoleteWorkersApi::class)
```

В целом эти API признаны устаревшими. Не предполагается ни дальнейшей доработки, ни развития, ни устранения проблем. Workers продолжают поддерживаться, но в итоге они будут заменены threads API, а затем признаны устаревшими. Не рекомендуется делать Workers публичными в общедоступном API и сильно на них завязываться. Разработчики Kotlin предлагают создать вместо этого специальную инкапсулирующую обертку, которая упростит работу и будущую миграцию на новый API.

4.6. Управление памятью

Swift и Objective-C используют автоматический подсчет ссылок (ARC). В Kotlin/Native есть собственный сборщик мусора, который также интегрирован с Objective-C/Swift ARC. Неиспользуемые объекты Kotlin автоматически удаляются. Вам не нужно предпринимать дополнительные шаги для управления временем жизни экземпляров Kotlin/Native с помощью Swift или Objective-C. Чтобы предотвращать утечки, мы дополнительно можем использовать слабые ссылки WeakReference. Повторим пример из главы 3 про общее и нативное.

Создадим для начала общую декларацию `expect` для класса (листинг 4.34).

Листинг 4.34. Expect-декларация слабой ссылки

```
expect class WeakReference<T : Any>(referred: T) {
  fun clear()
  fun get(): T?
}
```

На iOS используем `actual`, но с указанием `typeAlias` (листинг 4.35).

Листинг 4.35. Слабая ссылка для Kotlin Native

```
actual typealias WeakReference<T> = kotlin.native.ref.WeakReference<T>
```

Таким образом мы создаем привязку существующего типа к ссылке. В данном случае мы используем реализацию слабой ссылки из Kotlin Native.

Аналогично поступим с Android. Здесь мы используем слабую ссылку из Kotlin JVM (листинг 4.36).

Листинг 4.36. Слабая ссылка для Kotlin JVM

```
actual typealias WeakReference<T> = java.lang.ref.WeakReference<T>
```

4.7. Соединяем suspend с iOS

Для нативного Android-приложения какие-либо доработки при использовании асинхронных запросов из общей части не нужны. Используется практически идентичный API. А вот для iOS нам потребуется использовать некоторые приемы.

Мы уже знаем, что `suspend`-методы Kotlin конвертируются в методы с блоком обратного вызова `completion handler` или `async/await` на стороне iOS.

Также не стоит забывать, что в зависимости от поставленной задачи мы можем использовать не только обобщаемые методы. Код платформенных проектов в решении может быть неидентичен, поэтому нельзя исключать подключение сугубо нативной логики, вызовы которой мы можем сочетать с вызовами логики из общего модуля, что вполне оправданно и целесообразно. Поэтому разумно применить доступные подходы конкретной платформы.

Попробуем сделать наш Kotlin-код совместимым с `Combine Publishers`. Для этого превратим вызов `suspend`-функции в `AnyPublisher` с использованием `Future Deferred` и `Promise` (листинг 4.37).

Листинг 4.37. Использование Future Deferred-Promise для Combile Publishers

```
func getNewsList()-> AnyPublisher<[NewsItem], Error> {
       return Deferred {
           Future { promise in
               self.getNewsList { response, error in
                   if let data = response?.content?.articles {
                       promise(.success(data))
                   }
                   if let error = response?.errorResponse {
                       promise(.failure(CustomError(error: error.message)))
                   }
                   if let error = error {
                       promise(.failure(error))
```

```
                }
            }
        }

    }.eraseToAnyPublisher()
```

Для удобства можно даже вынести вызов метода в расширение extension-сервиса, осуществляющего запрос (листинг 4.38).

Листинг 4.38. Перенос кода в расширение

```
extension NewsService {

    func getNewsList()-> AnyPublisher<[NewsItem], Error> {
        return Deferred {
            Future { promise in
             //Вызов кода
                   }
        }
    }
```

Вызов `ObservableObject` (если мы говорим о SwiftUI) или другой части логики в коде будет абсолютно таким же, как и в других случаях работы с `Publishers` (листинг 4.39).

Листинг 4.39. Пример вызова созданного Publisher

```
// Вызов в ViewModel (ObservableObject)
func loadData() {
     let _ =  newsService.getNewsList().sink { result in
            switch result {
            case .failure(let error):
                print(error.localizedDescription)
            default:
                break
            }
        } receiveValue: { data in
            self.items = [NewsItem]()
            self.items.append(contentsOf: data)
        }.store(in: &store)

    }
```

Что ж, пока это выглядит как перенос решения для работы с `Combine`. Особого выигрыша не чувствуется, тем более если представить, что нам придется обернуть в `Publisher` каждый метод, который мы будем вызывать на стороне iOS. Попробуем теперь обобщить эту логику. Можно сделать своеобразный менеджер задач на стороне общего KMP-кода, но в большинстве случаев это банальный оверинжиниринг. Попробуем пойти через Kotlin Flows, которые мы можем представить в общем виде как `Flow<T>`.

4.8. Соединяем Kotlin Flow с iOS

В iOS в `Combine` есть специальные обертки свойств (property wrappers), которые позволяют подписываться на изменения их значений со стороны View. Для того чтобы подписаться на изменения параметров модели состояния (`ObservableObject`), их нужно пометить реактивным property wrapper `@Published` (листинг 4.40).

Листинг 4.40. Пример @Published

```
class NewsListVm: ObservableObject {
  // На это свойство можно подписаться
  @Published var news: [NewsItem] = []
}
```

В Kotlin для этих целей есть специальная конструкция Flow (потоки). Потоки асинхронны, реактивны и по умолчанию работают на холодном старте (листинг 4.41).

Листинг 4.41. Пример Flow

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking {
    // Создаем Flow
    val flow = flow {
        for (i in 1..3) {
            delay(1000) // Имитация асинхронной операции
            emit(i)     // Выдаем значение
        }
    }

    // Подписываемся на Flow
    flow.collect { value ->
        println("Received: $value")
    }
}
```

Flow поддерживает различные реактивные операции для преобразования значений. В основном в коде используются реализации `StateFlow` и `SharedFlow`. `StateFlow` — это поток данных, который всегда хранит текущее значение (состояние) и уведомляет подписчиков о его изменении. `SharedFlow` — это поток данных, который может эмитить несколько значений и поддерживать несколько подписчиков.

Давайте посмотрим, как с этим работать в Kotlin Multiplatform. На стороне shared-модуля в Kotlin работа с Flow выглядит так (листинг 4.42).

Листинг 4.42. Использование Flow в commonMain

```
//ViewModel
val newsFlow = MutableStateFlow<NewsList?>(null)
fun loadData() {
        scope.launch {
           val result =  newsService.getNewsList()
            newsFlow.value = result.content
        }
    }
```

Для Android получение данных с Flow идет нативно и просто, нам достаточно подписаться на событие (4.43).

Листинг 4.43. Пример подписки на стороне Android

```
lifecycleScope.launch {
   repeatOnLifecycle(Lifecycle.State.STARTED) {
     viewModel.newsList.collect {
       setupNews(it)
     }
   }
 }
```

Здесь вызов идет из компонента `ViewModel` Android, у которого есть свой собственный скоуп. Подписка же реализуется в UI-слое через механизм lifecycleScope.

Если `suspend`-методы на стороне iOS-приложения автоматически конвертируются либо в методы с `completionHandler`, либо в `async/await`, то с Kotlin Flow не все так просто. Прямого аналога для экспорта/интероп нет. Поэтому надо подписываться на Flow как на свойство, через оператор `collect`.

Для iOS `collect` — это `suspend`-функция, которая имеет свои обязательные параметры; одним из них является специальный коллектор `Kotlinx_coroutines_coreFlowCollector`, для Swift этот тип представляет собой протокол.

Для обработки Flow нам потребуется создать на стороне iOS-класс, имплементирующий протокол коллектора. Для удобства будем использовать псевдоним `Collector` с помощью typealias. Полученные значения будем передавать наружу через замыкание обратного вызова `callback`, вызывая его в переопределенном методе `emit` (листинг 4.44).

Листинг 4.44. Реализация обертки для подписки на Flow collector

```
// Объявление псевдонима типа
typealias Collector = Kotlinx_coroutines_coreFlowCollector

// Собственно класс-обертка для подписки
class Observer: Collector {
  let callback: (Any?) -> Void

  init(callback: @escaping (Any?) -> Void) {
    self.callback = callback
  }
```

```
  func emit(value: Any?, completionHandler: @escaping (KotlinUnit?, Error?) ->
                          Void) {
    // Передаем значение в callback
    callback(value)
    // Завершаем выполнение
    completionHandler(KotlinUnit(), nil)
  }
}
```

Теперь для подписки со стороны iOS создадим экземпляр нашего коллектора, к примеру, в `ViewModel` реализуемого экранного модуля. В completion-блок прокинем вызов методов обработки полученных значений (листинг 4.45).

Листинг 4.45. Создание экземпляра Observer и обработка данных

```
lazy var collector: Observer = {
  let collector = Observer { value in
    // Получаем данные из потока
    if let value = value as? NewsList {
      let data = value.articles
      // Выводим данные
      self.processNews(data: data)
    }
  }
  return collector
}()
```

Затем передадим наш коллектор как переменную в метод `collect` Flow (листинг 4.46).

Листинг 4.46. Подключение Observer к свойству

```
newsViewModel.newsFlow.collect(collector: self.collector,
                                          completionHandler: {_,_ in })
```

Удобнее всего будет сделать присваивание коллектора переменной при инициализации или вызове `ViewModel` из shared-модуля (листинг 4.47).

Листинг 4.47. Подключение к свойству Flow ViewModel

```
lazy var newsViewModel: NewsViewModel = {
        let newsViewModel =  NewsViewModel()
        newsViewModel.newsFlow.collect(collector: self.collector,
                                       completionHandler: {_,_ in })
        return newsViewModel
    }()
```

Это не единственный способ получить значения из Kotlin Flow в iOS-приложении. Мы можем создать класс-обертку в commonMain shared-модуля с собственным скоупом корутин. Этот класс позволит инкапсулировать подписку на Flow. Отдавать данные наружу будем так же, через completion-блок (листинг 4.48).

Листинг 4.48. Обертка вокруг Flow

```
class AnyFlow <T> (source: Flow <T> ): Flow <T> by source {

  fun collect(onEach: (T) -> Unit,
    onCompletion: ((cause: Throwable ? )-> Unit)? = null): Cancellable {

    // Скоуп работает внутри метода
    val scope = CoroutineScope(SupervisorJob() + Dispatchers.Main)

    scope.launch {
      try {
        // Подписываемся на результат Flow
        collect {
          onEach(it)
        }
        // Если нет ошибки, то просто вызываем завершение
        onCompletion(null)
      } catch (e: Throwable) {
        onCompletion?.invoke(e)
      }
    }

    return object: Cancellable {

      // Зачищаем scope, убираем подписку
      override fun cancel() {
        scope.cancel()
      }
    }
  }
}
```

В данном случае мы передаем два блока в метод `collect`: блок для обработки полученных значений и блок, который будет вызван при ошибке или исключении.

Создадим вспомогательный метод расширения для оборачивания `KotlinFlow` в `AnyFlow` (листинг 4.49).

Листинг 4.49. Расширение для конвертации Flow в AnyFlow

```
fun <T> Flow<T>.wrapToAny(): AnyFlow<T> = AnyFlow(this)
```

Полученное решение можно использовать в commonMain следующим образом (листинг 4.50):

Листинг 4.50. Пример работы с AnyFlow в commonMain или androidMain

```
val newsFlow = MutableStateFlow<NewsList?>(null)
val flowNewsItem =  newsFlow.wrapToAny()
```

Вызов на стороне iOS будет выглядеть более компактным (листинг 4.51). Достаточно вызвать метод `collect` у свойства типа `AnyFlow`.

Листинг 4.51. Работа с AnyFlow на стороне iOS

```
newsViewModel.flowNewsItem.collect { data in
     //Обработка данных
 } onCompletion: { error in
    //Обработка ошибок
  }
```

Если наше iOS-приложение использует подход `Combine` для работы, мы можем превратить вызовы нашего решения-обертки в `Publishers`. Добавим новый статический метод в расширение типа `Publishers` (листинг 4.52).

Листинг 4.52. Вызов AnyFlow в Publisher как расширение

```
extension Publishers {
    static func createPublisher<T>(
        wrapper: AnyFlow<T>
    ) -> AnyPublisher<T?, Error> {
        var job: shared.Cancellable? = nil
      //Kotlinx_coroutines_coreJob? = nil
        return Deferred {
            Future { promise in
                job = wrapper.collect(onEach: { value in
                    promise(.success(value))
                }, onCompletion: { error in
                    promise(.failure(CustomError (error:error)))
                })
            }.handleEvents( receiveCancel:
                                {
                job?.cancel()
            })
        }.eraseToAnyPublisher()
    }
}
```

Для этого будем использовать `Future-Promise` и `Deferred`. Добавим также обработку отмены Flow.

Мы можем создать и свой `Publisher` и спрятать все нюансы реализации в него (листинг 4.53).

Листинг 4.53. Специальный Publisher для работы с AnyFow

```
public struct FlowPublisher<T: AnyObject>: Publisher {
    public typealias Output = T
    public typealias Failure = Never

    private let wrapper: AnyFlow<Output>
    public init(wrapper: AnyFlow<Output>) {
        self.wrapper = wrapper
    }

    public func receive<S: Subscriber>(subscriber: S) where S.Input == Output,
                          S.Failure == Failure {
```

```
        let subscription = FlowSubscription(wrapper: wrapper,
                                            subscriber: subscriber)
        subscriber.receive(subscription: subscription)
    }

    final class FlowSubscription<S: Subscriber>: Subscription
where S.Input == Output, S.Failure == Failure {

        private var subscriber: S?
        private var job: shared.Cancellable? = nil

        private let wrapper: AnyFlow<Output>

        init(wrapper: AnyFlow<Output>, subscriber: S) {
            self.wrapper = wrapper
            self.subscriber = subscriber

            job = wrapper.collect(onEach: { data in
                subscriber.receive(data!)
            }, onCompletion: { error in
                if let error = error {
                    debugPrint(error.description())
                }
                subscriber.receive(completion: .finished)
            })
        }

        func cancel() {
            subscriber = nil
            job?.cancel()
        }

        func request(_ demand: Subscribers.Demand) {}
    }
}
```

Использовать код мы сможем следующим образом (листинг 4.54):

Листинг 4.54. Подключение Flow Publisher

```
func loadData() {
    let _ =  FlowPublisher<NewsList>(wrapper: newsViewModel.flowNewsItem)
.sink { result in
           // Закончили работу
        } receiveValue: { data in
           // Получили результат
        }.store(in: &store)
```

Как и в другом `Publisher`, для получения результата нам потребуется просто вызвать оператор `sink`.

Современный подход к разработке iOS-приложений предполагает реализацию работы с многопоточностью через `async/await`. Мы можем вызвать `suspend`-

метод shared-модуля напрямую как вариацию с обратным вызовом блоком `completionHandler`, обернуть в `withCheckedContinuation` и передать результат работы через вызов `continuation.resume` (листинг 4.55).

Листинг 4.55. Использование withCheckContinuation для создания async/await-метода

```
func loadNews() async-> Result<[NewsItem],Error> {
        return await withCheckedContinuation{ continuation in
            newsService.getNewsList(completionHandler: { response,
                                                          error in
                if let news = response?.content?.articles {
                    continuation.resume(returning: .success(news))
                }
                if let error = response?.errorResponse {
                    continuation.resume(returning:
                             .failure(CustomError(error: error.message)))
                }
                if let error = error {
                    continuation.resume(returning:
                .failure(CustomError(error: error.localizedDescription)))
                }
            })
        }
    }
```

Этот подход позволит перехватить исключения и добавить обработку ошибок. Кстати, примерно такой код мы и добавим в приложение в следующих главах.

Последние версии Kotlin поддерживают превращение `suspend` напрямую в `async/await`. Так что мы можем вызвать метод нашего сервиса как есть.

Чтобы поддержать работу с Flow через `async/await`, добавим общую функцию-обертку, которая отдаст итоговый результат через `await` (листинг 4.56). Так мы сможем совместить `AnyFlow` с `async/await`.

Листинг 4.56. Совмещаем async/await с AnyFlow

```
func requestAsync<T>(wrapper: AnyFlow<T>) async -> Result<T?,Error> {
        return await withCheckedContinuation{ continuation in
            wrapper.collect { result in
                continuation.resume(returning: .success(result))
            } onCompletion: { error in
                continuation.resume(returning:
                                    .failure(CustomError(error: error)))
            }
        }
    }
```

Вызов в коде iOS будет выглядеть так же элегантно, как примерно показано в листинге 4.57.

Листинг 4.57. Вызов requestAsync с передачей ему AnyFlow

```
@MainActor
func loadAndSetup() {
  Task {
    let result = await requestAsync(
      wrapper:
        newsViewModel.flowNewsItem)
    //... Магия какая-то
  }
}
```

Что ж, пока работа с Flow на стороне iOS-приложения выглядит замысловато. Возможно, в следующих версиях Kotlin интероп между Flow и Swift-кодом будет реализовываться автоматически и/или более элегантно.

ПОДВЕДЕМ ИТОГИ

- ✓ Основным инструментом для работы с многопоточностью в приложении KMP являются корутины.
- ✓ Современная реализация `Kotlinx Coroutines Core` берет на себя все затраты по реализации механизма `expect/actual` для подстановки инструментария под таргет. Например, создание `CoroutineDispatcher`.
- ✓ Всю работу с многопоточностью можно перенести в commonMain общего модуля.
- ✓ Работа с Flow на стороне iOS пока требует решений в виде собственных оберток.
- ✓ Внутренний API Kotlin/Native по работе с многопоточностью может измениться в одной из следующих версий.

4.9. Применение на практике

Не стоит забывать, что мы все-таки не просто изучаем теорию, а пишем приложение. Поэтому резюмируем, что нам потребуется для настройки многопоточности:

1. Создадим файл `Dispatchers` в commonMain и добавим туда диспетчеры корутин для работы в главном и фоновых потоках (листинг 4.58).

Листинг 4.58. Диспетчеры работы с потоками

```
val uiDispatcher: CoroutineDispatcher = Dispatchers.Main

val ioDispatcher: CoroutineDispatcher = Dispatchers.IO

val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
```

2. В `BaseViewModel` создадим `CoroutineScope` с использованием общего `uiDispatcher` (листинг 4.59).

Листинг 4.59. Скоуп в базовой ViewModel

```
class BaseViewModel(val dispatcher = uiDispatcher) {

  val scope: CoroutineScope = CoroutineScope(dispatcher)

  // Код и логика
}
```

Нам может потребоваться вызывать бизнес-логику в фоне. Для этого будем использовать `UseCase`. В базовый класс `BaseUseCase` добавим обертку выполнения в контекст фоновой корутины (листинг 4.60).

Листинг 4.60. Код базового UseCase

```
import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.withContext

abstract class BaseUseCase<in T, out R>(
        private val dispatcher: CoroutineDispatcher = ioDispatcher
) {

  abstract suspend fun execute(param: T): R

  suspend operator fun invoke(param: T): Result<R> = withContext(dispatcher) {
    runCatching { execute(param) }
  }
}
```

Также мы добавили перехват ошибок и возврат в формате Result.

3. В `NewsViewModel` добавим свойство для считывания информации по получаемым данным (листинг 4.61).

Листинг 4.61. Свойство для списка новостей

```
val newsFlow = MutableStateFlow<NewsList?>(null)
```

4. Далее вы можете выбрать один из вариантов реализации (см. код выше):
   - создать специальную обертку для Flow на стороне Kotlin Common и подключить на стороне iOS через `completionHandler`;
   - создать обертку для `Coroutines Flow Collector` на стороне iOS (см. код выше, листинги 4.45–4.47).

Мы реализовали полноценную заготовку нашего приложения. Теперь пора подключить сетевой клиент и наполнить его настоящей логикой.

В следующей главе мы рассмотрим, как реализовать сетевой слой нашего приложения с многопоточной работой. Будем использовать `Dispatchers`, а также обертки для Flow. Кроме того, нам нужно будет соединить сетевой слой с другими частями нашего приложения.

ГЛАВА 5

Сетевой слой приложения KMP

В этой главе рассмотрим:

- реализацию сетевых клиентов для Android и iOS в общем модуле;
- написание общего сетевого слоя.

5.1. Подготовка к работе

Ни одно современное мобильное приложение не обходится без сетевого слоя. Пришло время наполнить и наше приложение жизнью. Мы будем отображать реальные данные, которые получим от стороннего API news.api.org по запросу (рис. 5.1).

Data-классы для ответа у нас уже есть, мы их заполнили в главе 2 (см. листинг 2.1), с дорожной картой и постановкой задач для нашего приложения определились. Продублируем их на всякий случай (листинг 5.1).

Листинг 5.1. Модели для ответов API

```
// Source
data class Source(
  val id: String,
  val name: String,
)

// Модель для новости
data class NewsItem(
  val source: Source? = null,
  val author: String,
  val title: String,
  val description: String,
  val url: String? = null,
  val publishedAt: String? = null,
  val content: String,
  val urlToImage: String? = null
)

// Список новостей
```

```
data class NewsItemsList(
  val totalResults: Int,
  val articles: List<NewsItem>
)
```



Рис. 5.1. Пример API

Данные нам будут приходить в виде JSON. Для того чтобы мы могли их десериализовать в модели, нужно подключить специальный плагин для сериализации/десериализации kotlinx.serialization (https://kotlinlang.org/docs/serialization.html). Пропишем настройки подключения в `build.gradle.kts` для нашего приложения (листинг 5.2).

Листинг 5.2. Подключение плагина kotlinx.serialization

```
plugins {
    //trick: for the same plugin versions in all sub-modules
    //…
```

```
    kotlin("plugin.serialization").version(*).apply(false)
}
```

Также можно использовать нотацию с Gradle-каталогом библиотек (листинг 5.3).

Листинг 5.3. Подключение плагина сериализации через каталог Gradle

```
plugins {
    //trick: for the same plugin versions in all sub-modules
    alias(libs.plugins.serialization).apply(false)
}
```

Пути к библиотеке и версию в таком случае надо будет указать в `gradle/libs.versions` (листинг 5.4)

Листинг 5.4. Пример добавления библиотеки в каталог

```
[versions]
 kotlin = "2.0.0"
  kotlinSerialization = "2.1.0"


[plugins]
serialization = {id= "org.jetbrains.kotlin.plugin.serialization", version.ref =
                    "kotlinSerialization"}
```

и блоке plugins в `build.gradle.kts` общего модуля (листинг 5.5).

Листинг 5.5. Подключение плагина в build.gradle.kts commonMain

```
plugins {
    //...
    kotlin("plugin.serialization")
}
```

Укажем версию, совпадающую с текущей версией Kotlin в нашем приложении или совместимую, но не ниже. Также добавим зависимость для commonMain (листинг 5.6).

Листинг 5.6. Добавление сериализации в блок зависимостей

```
repositories {
  mavenCentral()
}

commonMain.dependencies {
  implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:*")
}
```

Запустим синхронизацию Gradle для установки.

Теперь нужно включить поддержку сериализации в наших data-классах. Для этого воспользуемся аннотацией `@Serializable` и отметим ею весь класс (листинг 5.7).

Листинг 5.7. Настройка сериализации для data-класса

```
@kotlinx.serialization.Serializable
data class NewsItem(
        val source: Source? = null,
        @SerialName("author") val author: String?,
        val title: String,
        val description: String,
        val url: String? = null,
        val publishedAt: String? = null,
        val content: String,
        val urlToImage: String? = null
)
```

Аннотация `@SerialName` может использоваться, если имя свойства класса и имя поля JSON отличаются по тем или иным причинам. Если у вас использовался Gson, то надо будет заместить этой аннотацией `@Expose` и `@Serialized`.

Если вы мигрируете с нативного приложения Android, то изначально у вас используется протокол `java.io.Serializable` (листинг 5.8).

Листинг 5.8. Исходная модель данных в Android

```
import java.io.Serializable

data class NewsItem(
        val source: Source? = null,
        @Expose
        @Serialized("author")
        val author: String?,
        val title: String,
        val description: String,
        val url: String? = null,
        val publishedAt: String? = null,
        val content: String,
        val urlToImage: String? = null
) : Serializable
```

При миграции с iOS и Swift обратите внимание на следующее. Здесь нет аналогов `enum CodingKeys` с перечнем нужных полей (листинг 5.9).

Листинг 5.9. Исходная модель данных в iOS

```
import Foundation

struct NewsItem: Codable, Identifiable, Hashable {
  var id: UUID = UUID()

  let author: String?
  let title: String?
  let _description: String?
  let content: String?
  let urlToImage: String?
  let publishedAt: String?
```

```
// Эти поля нам нужны
  enum CodingKeys: String, CodingKey {
    case author = "author"
    case title = "title"
    case _description = "description"
    case urlToImage = "urlToImage"
    case publishedAt = "publishedAt"
    case content = "content"
  }
}
```

Если вам не нужно какое-то поле в Kotlin, то достаточно просто сделать его опциональным.

Перейдем, собственно, к сетевому клиенту и его заготовке.

Для удобства работы мы сразу разместили служебные данные для сетевых запросов в отдельный файл конфигурации (листинг 5.10).

Листинг 5.10. Пример файла конфигурации для сетевого клиента

```
class NetworkConfiguration {
  companion object {
    const val BASE_URL = "https://newsapi.org/v2/"
    const val API_KEY = "**********"
  }
}
```

Конфигурацию в нашу заготовку сетевого клиента можно передать через конструктор (листинг 5.11).

Листинг 5.11. Пример подключения конфигурации к сетевому клиенту

```
class NetworkClient constructor(
        private val networkConfiguration: NetworkConfiguration
) { }
```

Другой вариант — запрашивать параметры напрямую, так как в нашем случае это реализуется через статический доступ.

Теперь займемся логикой сетевых запросов. Рассмотрим последовательно следующие решения:

1) использование готовой библиотеки (Ktor);
2) написание собственных сетевых клиентов в shared-модуле.

Основным решением для работы с сетевыми запросами в приложениях KMP является использование кросс-платформенного фреймворка Ktor — одной из первых кросс-платформенных библиотек. Эта библиотека с открытым исходным кодом, созданная и поддерживаемая JetBrains, а также сообществом. Есть версии как для клиентских, так и для серверных приложений. Эта библиотека исполь-

зовалась с самыми первыми версиями SDK KMP, еще до выхода alpha-версии, поэтому ее по праву можно назвать родным и даже «нативным» KMP-решением.

Ktor позволяет реализовать общий сетевой слой в общем shared-модуле. Его легко подключить, легко настроить и просто использовать. Внутри в нативных компонентах модуля библиотека Ktor использует обращение к нативному механизму сетевых запросов Android SDK и iOS SDK. Для того чтобы разобраться, как это работает, мы начнем с реализации сетевого слоя на стандартных нативных (насколько это возможно для iOSMain и androidMain) компонентах.

5.2. Нативная реализация в commonMain

Начнем с раздельной реализации сетевых клиентов, как если бы мы придерживались вот такой архитектуры (рис. 5.2).

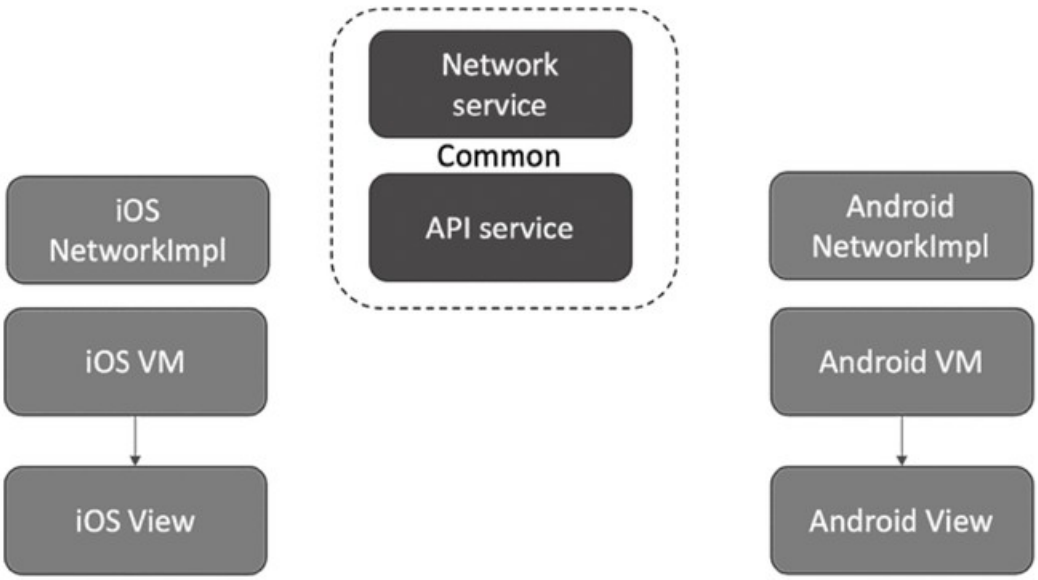


Рис. 5.2. Схема раздельной архитектуры и сетевых клиентов

Затем перейдем к переносу и адаптации кода в shared-модуль на Kotlin.

На Android сетевой слой реализуется обычно с использованием библиотеки OkHttp, для удобства и простоты реализации дополнительно можно использовать Retrofit. На iOS используется Network Framework из iOS SDK. Пример сетевого клиента iOS представлен в листинге 5.12.

Листинг 5.12. Сетевой клиент на Network Framework

```
class NetworkClient {

  // Создание Url-сессии
  private var urlSession: URLSession = {
```

```
    return URLSession(configuration: .default)
  }()

  private var dataTask: URLSessionDataTask? = nil

  func request<T: Codable>(path: String, completion: @escaping (Result<T,
Error>) -> Void) {
    // Создаем NSUrl по строковому пути
    guard let url = URL(string: path) else {
      return completion(.failure(CustomError(message: "Bad wrong url")))
    }

    // Создаем URLRequest
    var urlRequest = URLRequest(url: url)
    // Добавляем заголовки из конфигурации
    for h in networkConfiguration.getHeaders() {
      urlRequest.setValue(h.value, forHTTPHeaderField: h.key)
    }

    // Запуск задачи на загрузку
    self.dataTask = urlSession.dataTask(
      with: urlRequest,
      completionHandler: { data, response, error in
        if let error = error {
          DispatchQueue.main.async {
            completion(.failure(error))
          }
        }
        if let data = data {
          // Десериализация
          do {
            let content = try self.jsonDecoder.decode(T.self, from: data)
            DispatchQueue.main.async {
              completion(.success(content))
            }
          } catch {
            DispatchQueue.main.async {
              completion(.failure(error))
            }
          }
        }
      })

    self.dataTask?.resume()
  }
}
```

Решение для Android легко перенести в androidMain, поэтому начнем с адаптации клиента для iOS.

Напишем версию на Kotlin Native в iOSMain (листинг 5.13).

Листинг 5.13. Сетевой клиент для iOS на Kotlin Native

```
actual class NetworkHttp {

  // NSURL-сессия
  val urlSession: NSURLSession by lazy {
    NSURLSession.sharedSession
  }

  //Метод запроса
  actual suspend inline fun<reified T> request(path: String): Result<T> {
    //Создаем URL из строки
    val url = NSURL.URLWithString(path)
    val headers = NetworkConfig.header as Map<Any?, *>
    // Создаем URLRequest из URL
    val urlRequest = NSMutableURLRequest(url!!)
    // Добавляем заголовки
    urlRequest.setAllHTTPHeaderFields(headers)
    // Загрузка данных
    return suspendCoroutine<Result<T>> { coroutine ->
      urlSession.dataTaskWithRequest(urlRequest) { data,
      response, error ->

        try {
          val stringData = NSString.create(data!!, encoding =
NSUTF8StringEncoding).toString()
          //Десериализация
          val content = JsonDecoder.instance.decode<T>(stringData)
          coroutine.resume(Result.success(content))
        } catch (e: Exception) {
          coroutine.resume(Result.failure(e))
        }
      }.resume()
    }
  }
}
```

Обратите внимание: нужные нам библиотеки доступны через `platform.Foundation` (листинг 5.14).

Листинг 5.14. Примеры библиотеки Kotlin Native из platform.Foundation

```
import platform.Foundation.NSMutableURLRequest
import platform.Foundation.NSString
import platform.Foundation.NSURL
import platform.Foundation.NSURLSession
import platform.Foundation.NSUTF8StringEncoding
import platform.Foundation.create
import platform.Foundation.dataTaskWithRequest
import platform.Foundation.setAllHTTPHeaderFields
import kotlin.coroutines.resume
import kotlin.coroutines.suspendCoroutine
```

Это почти то же самое, что мы писали нативно на Swift. Решение немного упростилось в части работы с `NSURLSession`, это связано с особенностями реализации портирования на Kotlin Native.

Мы также превратили вызов метода `dataTaskWithRequest` в корутину, обернув его в `suspendCoroutine` и используя метод `resume` у `continuation` корутины. Это практически аналогично тому, как если бы мы использовали на стороне iOS обертку `withCheckedContinuation` из механизма `async/await`.

Для упрощения десериализации можно написать вот такой класс-помощник на базе `kotlinx.serialization` и расположить его в commonMain (листинг 5.15).

Листинг 5.15. Десериализатор для Kotlin Native

```
class JsonDecoder{
  companion object {
    val instance = JsonDecoder()
    val jsonDecoder = kotlinx.serialization.json.Json { ignoreUnknownKeys = true
}
  }

  inline fun<reified T> decode(json:String): T {
    return jsonDecoder.decodeFromString(json)
  }
}
```

Обратите внимание, что наш клиент для iOS мы сразу пометили `actual`, потому что нам нужно связать его с `expect`-декларацией в commonMain (листинг 5.16).

Листинг 5.16. Декларация expect сетевого клиента

```
expect class NetworkHttp {
  suspend inline fun<reified T> request(path: String): Result<T>
}
```

Теперь добавим простую реализацию на OkHttp для Android в androidMain (листинг 5.17).

Листинг 5.17. Сетевой клиент для androidMain

```
actual class NetworkHttp {
  val client: OkHttpClient by lazy {
    OkHttpClient()
  }

  actual suspend inline fun<reified T> request(path: String): Result<T> {
    val requestBuilder = Request.Builder()
    // Добавляем в запрос url
    .url(path)

    //Добавляем заголовки
    for (header in NetworkConfig.header) {
      requestBuilder.addHeader(header.key, header.value)
```

```
    }
    requestBuilder.method("GET",null)

    val request = requestBuilder.build()
    try {
      // Инициализируем асинхронный вызов
      val response = client.newCall(request).await()
      // Десериализуем
      val content = JsonDecoder.instance.decode<T>(response.body!!.string())
      return Result.success(content)
    } catch (e: Exception) {
      return Result.failure(e)
    }
  }
}
```

Такой код будет работать и в нативном Android-приложении.

Добавим зависимости для androidMain в `build.gradle.kts` (листинг 5.18).

Листинг 5.18. Подключение зависимостей в androidMain

```
androidMain.dependencies {

  implementation("com.squareup.okhttp3:okhttp:*")
  implementation("com.squareup.okhttp3:okhttp-bom:*")

  // define any required OkHttp artifacts without version
  implementation("com.squareup.okhttp3:okhttp")
  implementation("com.squareup.okhttp3:logging-interceptor")
}
```

Для создания запроса мы использовали метод `newCall` OkHttpClient. Чтобы превратить обработку вызова в корутину, можно использовать самописный `callback` и расширение (листинг 5.19).

Листинг 5.19. Класс-обертка для превращения Callback ответа OkHttpClient в корутину

```
class ContinuationCallback(
        private val call: Call,
        private val continuation: CancellableContinuation<Response>
) : Callback, CompletionHandler {

    override fun onResponse(call: Call, response: Response) {
        // Передаем результат в continuation
        continuation.resume(response)
    }

    override fun onFailure(call: Call, e: IOException) {
        if (!call.isCanceled()) {
            continuation.resumeWithException(e)
        }
    }
```

```
    override fun invoke(cause: Throwable?) {
        try {
            call.cancel()
        } catch (_: Throwable) {}
    }
}

// Расширение для вызова ContinuationCallback в рамках suspendCoroutine
suspend inline fun Call.await(): Response {
    return suspendCancellableCoroutine { continuation ->
        val callback = ContinuationCallback(this, continuation)
        enqueue(callback)
        continuation.invokeOnCancellation(callback)
    }
}
```

Как и в примере с iOS, мы использовали обертку `suspendCoroutine`, но в cancellable-варианте и прокинули передачу результата или ошибки в `resume continuation` корутины внутри методов нашего блока обратного вызова.

Также для Android-приложения мы можем использовать сетевой клиент на `Retrofit` (листинг 5.20).

Листинг 5.20. Пример реализации Retrofit

```
object RetrofitClient {

  private const val BASE_URL = "https://newsapi.org/v2/"

  // Инстанс сервиса с использованием Retrofit
  val api: NewsApiService by lazy {
    Retrofit.Builder()
      .baseUrl(BASE_URL)
      .addConverterFactory(GsonConverterFactory.create())
      .build()
      .create(NewsApiService::class.java)
  }
}
```

Для установки Retrofit достаточно добавить зависимость в `androidMain.dependencies`, указав актуальную версию, как в листинге 5.21.

Листинг 5.21. Подключение Retrofit

```
androidMain.dependencies {`
         implementation("com.squareup.retrofit2:retrofit:*")
}
```

Проверим результат на iOS (рис. 5.3) и на Android (рис. 5.4).

Получилось простое рабочее решение, которое вполне можно использовать, если потребуется по-разному реализовать сетевые клиенты на Android и iOS или есть необходимость в дополнительном функционале, например в сокетах или SSL.



Рис. 5.3. Пример работы приложения на iOS

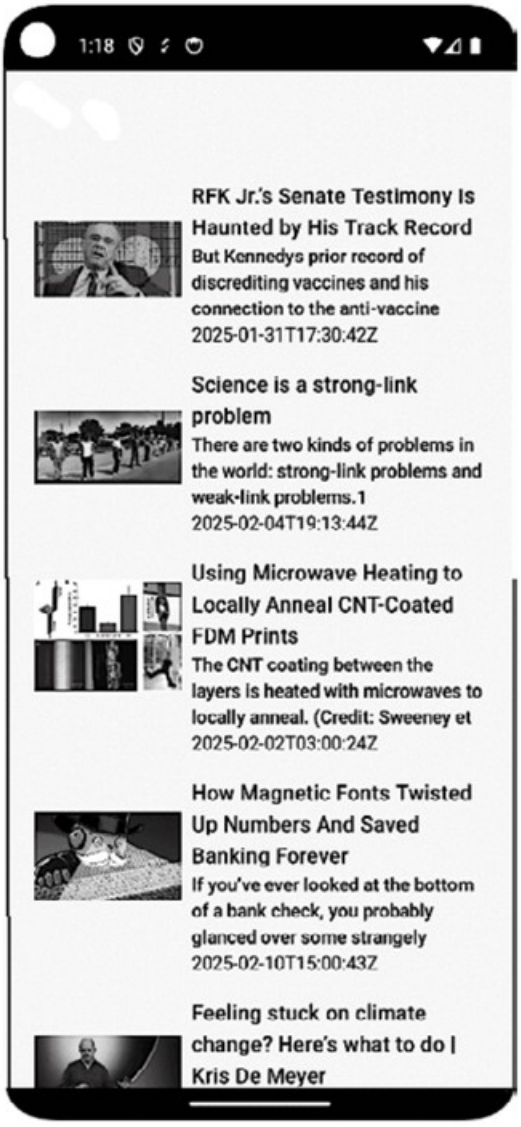


Рис. 5.4. Пример работы приложения на Android

5.3. Общий сетевой клиент на Ktor

Теперь попробуем сделать общий сетевой клиент на Ktor. Начнем с установки библиотек. Нам потребуется добавить зависимости в `build.gradle.kts` shared-модуля (листинг 5.22). Информацию об актуальной версии и точный путь берем в документации: https://ktor.io/docs/client-create-multiplatform-application.html.

Листинг 5.22. Подключение Ktor к commonMain

```
sourceSets {
    commonMain.dependencies {
        implementation("io.ktor:ktor-client-core:*")
        }
    }
```

Вместо звездочки указываем нужную версию. Обратите внимание, что версия Ktor должна соотноситься с версией Kotlin. Например, на момент написания книги это Kotlin 2.0 и Ktor 2.3.12.

Внутри в нативных компонентах модуля библиотека Ktor использует обращение к нативному механизму сетевых запросов Android SDK и iOS SDK. Поэтому добавим соответствующие блоки зависимостей для androidMain и iOSMain (листинг 5.23).

Листинг 5.23. Подключение Ktor к androidMain и iOSMain

```
androidMain.dependencies {
    implementation("io.ktor:ktor-client-okhttp:*")
}

iosMain.dependencies {
  implementation("io.ktor:ktor-client-darwin:*")
}
```

Для настройки и конфигурации engine сетевых клиентов для iOSMain и androidMain нам потребуется добавить зависимости для логирования и десериализации (листинг 5.24).

Листинг 5.24. Оставшиеся зависимости Ktor

```
implementation("io.ktor:ktor-client-logging:*")
implementation("io.ktor:ktor-client-content-negotiation:*")
implementation("io.ktor:ktor-client-serialization:*")
implementation("io.ktor:ktor-serialization-kotlinx-json:*")
```

Мы уже подключили поддержку сериализации в приложении как плагин, а также добавили соответствующий код к data-классам. Данные зависимости нужны именно для Ktor.

Нажимаем кнопку синхронизации, и мы почти готовы приступать к работе. Пришло время заполнить кодом нашу заготовку сетевого клиента, как на рис. 5.5.

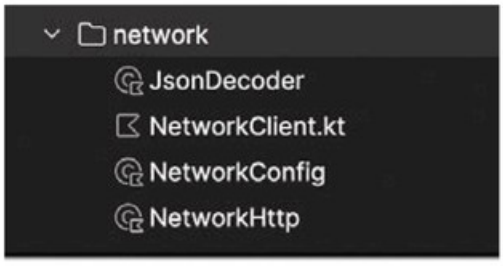


Рис. 5.5. Примерная иерархия решения

В файле `NetworkClient` задекларируем как `expect` глобальный метод для создания `HttpClient` (листинг 5.25). Это тот компонент, который мы будем использовать под капотом сетевого слоя нашего приложения.

Листинг 5.25. Декларация сетевого клиента в commonMain

```
expect fun createHttpClient() : HttpClient
```

В androidMain и iOSMain создадим `actual`-реализации `HttpClient` (листинг 5.26).

Листинг 5.26. HttpClient для androidMain

```
//Реализация actual HttpClient
actual fun createHttpClient(): HttpClient {
  return AndroidClient()
}

// Используем движок OkHttp
fun AndroidClient() = HttpClient(OkHttp) {
  // Настройки JSON
  install(ContentNegotiation) {
    json(Json {
      prettyPrint = true
      ignoreUnknownKeys = true
      explicitNulls = true
    })
  }
  // Настройки логирования
  install(Logging) {
    logger = Logger.DEFAULT
    level = LogLevel.ALL
  }
}
```

Создадим класс для http-клиента Android. Укажем нужный нам движок, а также настройки для работы с Json и ключами.

`HttpClient` для Android работает на движке сетевого клиента OkHttp.

Создадим теперь аналогичный класс HttpClient для iOS (листинг 5.27).

Листинг 5.27. HttpClient для iOS на движке Darwin

```
// Реализация actual HttpClient
actual fun createHttpClient(): HttpClient {
  return iOSClient()
}

// Используем движок Darwin
fun iOSClient() = HttpClient(Darwin) {
  // Настройки JSON
  install(ContentNegotiation) {
    json(Json {
      prettyPrint = true
      ignoreUnknownKeys = true
      explicitNulls = true
    })
  }
```

```
  // Настройки логирования
  install(Logging) {
    logger = Logger.DEFAULT
    level = LogLevel.ALL
  }
}
```

Единственное отличие от Android-версии — это использование движка `Darwin`.

Соберем все вместе: добавим также логику обработки запросов в общем виде в `NetworkClient` (листинг 5.28).

Листинг 5.28. Сетевой клиент Ktor

```
expect fun createHttpClient(): HttpClient

class NetworkClient {
  // Вызов создания HttpClient для таргета
  private val httpClient = createHttpClient()
  // Метод запроса
  suspend inline fun <reified T> request(path: String): Result<T> {
    return try {
      // GET-запрос
      val data = httpClient.get(path)
      // print(data) < - - Просто вывод
      // Десериализация
      val result = data.body<T>()
      Result.success(result)
    } catch (e: Exception) {
      Result.failure(e)
    }
  }
}
```

Реализации метода `createHttpClient` подтянутся автоматически для каждого таргета, а также и все нужные движки. Инициализируем переменную для `HttpClient`.

Для запросов в общем виде создадим generic-метод `request`, который вернет нам ответ в виде `Result.success` или `failure`. Так как мы используем корутины в качестве механизма многопоточности, то и метод у нас будет `suspend`. Если вы iOS-разработчик, то, вероятно, у вас могут вызвать вопросы ключевые слова `inline reified`. Они нужны для того, чтобы сохранить информацию о типе параметра или ответа и передать ее дальше в рантайм.

Метод `httpClient.get` позволяет нам выполнить GET-запрос и получить `HttpResponse`. Преобразование ответа в нужный тип данных производится автоматически с помощью вызова `body` у `HttpResponse` и указания типа: `data.body<T>()`.

На любом из этапов мы можем получить исключение, поэтому обернем код в `try-catch` и вернем `Result.failure` в случае ошибки.

Доработаем наш код и добавим обработку других запросов (листинг 5.29). В качестве параметра для указания типа метода и выбора команды используем простой enum.

Листинг 5.29. Расширенная версия сетевого клиента Ktor

```
enum class Method {
  GET,
  POST,
  PUT,
  DELETE
};

suspend inline fun <reified T> request(
    path: String,
    method: Method,
    body: Any? = null,
): Result<T> {
  val url = "${NetworkConfig.apiUrl}$path"
  return try {
    // Вызов метода в зависимости от ключа
    val data = when (method) {
      Method.GET -> httpClient.get(url) {
        // Добавляем заголовки
        NetworkConfig.header.forEach {
          header(it.key, it.value)
        }
      }
      Method.POST -> httpClient.post(url) {
        contentType(ContentType.Application.Json)
        // Кладем body
        setBody(body)
        // Добавляем заголовки
        NetworkConfig.header.forEach {
          header(it.key, it.value)
        }
      }
      Method.PUT -> {…}
      Method.DELETE -> {…}
    }
    // Получение данных и десериализация
    val result = data.body<T>()
    Result.success(result)
  } catch (e: Exception) {
    Result.failure(e)
  }
}
```

Вызовем у httpClient подходящий метод: get, post, put, delete. Также добавим заголовок из NetworkConfig в качестве header. Body просто передадим как некий Any-объект. Для Put и Delete код аналогичен.

В целом такого решения более чем достаточно для сетевого слоя, если нет потребности в сокетах или SSL.

5.4. Материал со звездочкой. Внутренняя кухня Ktor

Теперь давайте заглянем под капот наших `HttpClient`. Начнем с Android-клиента, работающего на движке от OkHttp (листинг 5.30).

Листинг 5.30. Фрагмент кода движка OkHttp

```
A JVM/Android client engine that uses the OKHttp HTTP client. This engine
supports Android 5.0 and newer.

To create the client with this engine, pass it to the HttpClient constructor:
val client = HttpClient(OKHttp)

To configure the engine, pass settings exposed by OkHttpConfig to the engine
method:
val client = HttpClient(OKHttp) {
  engine {
    // this: OkHttpConfig
  }
}

You can learn more about client engines from Engines.
public object OkHttp : HttpClientEngineFactory<OkHttpConfig> {
override fun create(block: OkHttpConfig.() -> Unit): HttpClientEngine =
             OkHttpEngine(OkHttpConfig().apply(block))
}
```

Посмотрим на интерфейс `HttpClientEngine` (листинг 5.31).

Листинг 5.31. Фрагмент кода HttpClientEngine

```
public interface HttpClientEngine : CoroutineScope, Closeable {

  /**
   * Specifies [CoroutineDispatcher] for I/O operations.
   */
  public val dispatcher: CoroutineDispatcher

  /**
   * Provides access to an engine's configuration.
   */
  public val config: HttpClientEngineConfig

  /**
   * Set of supported engine extensions.
   */
```

```
  public val supportedCapabilities: Set<HttpClientEngineCapability<*>>
    get() = emptySet()

  private val closed: Boolean
    get() = !(coroutineContext[Job]?.isActive ?: false)

  /**
   * Creates a new [HttpClientCall] specific for this engine, using
     a request [data].
   */
  @InternalAPI
  public suspend fun execute(data: HttpRequestData): HttpResponseData
  //...
}
```

Обратите внимание, что этот интерфейс декларирует свойства под диспетчера и контекст корутин. Если посмотреть на базовый класс `HttpClientEngineBase` (листинг 5.32), то можно заметить, что используется `Dispatchers.IO`.

Листинг 5.32. Фрагмент кода HttpClientEngineBase

```
/**
 * Abstract implementation of [HttpClientEngine] responsible for lifecycle
   control of [dispatcher] and
 * [coroutineContext] as well as proper call context management. Should be
   considered as the best parent class for
 * custom [HttpClientEngine] implementations.
 */
public abstract class HttpClientEngineBase(private val engineName: String) :
                      HttpClientEngine {
    private val closed = atomic(false)

    override val dispatcher: CoroutineDispatcher = ioDispatcher()

    override val coroutineContext: CoroutineContext by lazy {
        SilentSupervisor() + dispatcher + CoroutineName("$engineName-context")
    }

    override fun close() {
        if (!closed.compareAndSet(false, true)) return
        val requestJob = coroutineContext[Job] as? CompletableJob ?: return
        requestJob.complete()
    }
}
```

Это значит, что Ktor берет на себя ответственность за корректное асинхронное выполнение ваших сетевых запросов. Функционал такого рода существовал даже в ранних версиях библиотеки. До полноценной адаптации многопоточности iOS/Kotlin Native это было особенно удобно, так как позволяло полноценно выполнять запросы в фоне.

Теперь посмотрим на внутренний код движка `Darwin` (листинг 5.33).

Листинг 5.33. Код движка Darwin

```
@OptIn(InternalAPI::class)
public object Darwin : HttpClientEngineFactory<DarwinClientEngineConfig> {
  init {
    engines.append(this)
  }

  override fun create(block: DarwinClientEngineConfig.() -> Unit):
  HttpClientEngine = DarwinClientEngine(DarwinClientEngineConfig().apply(block))

  override fun toString(): String = "Darwin"
}
```

Приглядимся внимательнее к устройству `DarwinClientEngine` (листинг 5.34).

Листинг 5.34. Фрагмент кода DarwinClientEngine

```
@OptIn(InternalAPI::class)
internal class DarwinClientEngine(override val config:
               DarwinClientEngineConfig) :
               HttpClientEngineBase("ktor-darwin") {

  private val requestQueue: NSOperationQueue? = when (val queue =
                            NSOperationQueue.currentQueue()) {
    NSOperationQueue.mainQueue -> NSOperationQueue()
    else -> queue
  }
  override val dispatcher = Dispatchers.Unconfined
  override val supportedCapabilities = setOf(HttpTimeout, WebSocketCapability)

  private val session = DarwinSession(config, requestQueue)

  override suspend fun execute(data: HttpRequestData): HttpResponseData {
    val callContext = callContext()
    return session.execute(data, callContext)
  }
}
```

Для запросов используется `DarwinSession` — инкапсулированная логика, работающая с `NSUrlSession`. Также для маршрутизации запросов в фоновый поток здесь используется `NSOperationQueue`.

Заглянем внутрь `DarwinSession` (листинг 5.35).

Листинг 5.35. Фрагмент кода DarwinSession

```
@OptIn(UnsafeNumber::class)
internal class DarwinSession(
        private val config: DarwinClientEngineConfig,
        requestQueue: NSOperationQueue?
) : Closeable {
    private val closed = atomic(false)
```

```
    private val sessionAndDelegate = config.sessionAndDelegate ?:
                createSession(config, requestQueue)
    private val session = sessionAndDelegate.first
    private val delegate = sessionAndDelegate.second
```

`DarwinClientEngineConfig` может содержать данные о созданной сессии `NSUrlSession` и делегате `KtorNSURLSessionDelegate`. Если таких данных в конфигурации нет, то создадим их с помощью метода `createSession`, как в листинге 5.36.

Листинг 5.36. Пример создания сессии в DarwinClientEngine

```
@OptIn(UnsafeNumber::class)
internal fun createSession(
        config: DarwinClientEngineConfig,
        requestQueue: NSOperationQueue?
): Pair<NSURLSession, KtorNSURLSessionDelegate> {
    val configuration = NSURLSessionConfiguration.defaultSessionConfiguration().
apply {
        setupProxy(config)
        setHTTPCookieStorage(null)

        config.sessionConfig(this)
    }
    val delegate = KtorNSURLSessionDelegate(config.challengeHandler)

    return NSURLSession.sessionWithConfiguration(
            configuration,
            delegate,
            delegateQueue = requestQueue
    ) to delegate
}
```

В целом код очень похож на iOS-нативную реализацию. Обратите внимание, что для сбора данных внутри будет использоваться не `completionHandler`, а делегат `NSURLSessionDelegate`.

Теперь рассмотрим основной метод сессии execute (листинг 5.37). Сначала создается `Request`, потом происходят настройка и выполнение задачи. Используется механизм для `await` данных из делегата (мы видели что-то подобное в главе 4 о многопоточности). Запуск задачи производится через `task.resume`.

Листинг 5.37. Выполнение задачи

```
@OptIn(InternalAPI::class)
internal suspend fun execute(request: HttpRequestData, callContext:
CoroutineContext): HttpResponseData {
  val nativeRequest = request.toNSUrlRequest()
      .apply(config.requestConfig)
  val (task, response) = if (request.isUpgradeRequest()) {
    val task = session.webSocketTaskWithRequest(nativeRequest)
    val response = delegate.read(task, callContext)
    task to response
  } else {
```

```
    val task = session.dataTaskWithRequest(nativeRequest)
    val response = delegate.read(request, callContext, task)
    task to response
  }

  task.resume()

  try {
    return response.await()
  } catch (cause: Throwable) {
    if (task.state == NSURLSessionTaskStateRunning) task.cancel()
    throw cause
  }
}
```

Если бы мы писали код сами, у нас получилось бы что-то подобное, за исключением синтаксиса и дополнительных параметров. Механизм считывания ответа инкапсулирован внутри делегата `KtorNSURLSessionDelegate`, который представляет собой, как и в случае с `DarwinSession`, обертку вокруг реализации логики `NSURLSessionDelegate`.

Больше и подробнее о Ktor вы можете прочитать в документации на официальном сайте https://ktor.io/.

ПОДВЕДЕМ ИТОГИ

- ✓ KMP и Kotlin Native поддерживают достаточный функционал iOS SDK для реализации практически нативного сетевого клиента в shared-модуле.
- ✓ Если нужно сделать общий и максимально простой сетевой слой, то вполне подойдет Ktor.

5.5. Применение на практике

Давайте теперь дополним наш сетевой слой.

1. Добавьте зависимости для `kotlinx.serialization`.
2. Если вы еще не создали data-классы для ответов нашего API, то воспользуйтесь листингом 5.1.
3. Добавьте зависимости для Ktor (листинги 5.22–5.24). Не забывайте нажимать на кнопку синхронизации Gradle.
4. Реализуйте `HttpClient`, как было показано в листингах 5.25–5.27.
5. Добавьте сетевой клиент из листингов 5.28 или 5.29.

В следующей главе мы пройдемся по контрольным точкам и соединим вместе реализованные компоненты приложения. Если вы начали работу с книгой с разделов, посвященных практике, то не забудьте ознакомиться и с теорией.

ГЛАВА 6

Собираем приложение по частям

В этой главе рассмотрим:

- как собрать приложение из каркаса и полезных сниппетов;
- как выполнить миграцию на `ViewModel` для Kotlin Multiplatform от Jetpack Compose.

Эта глава является сугубо практической и предназначена для сведения воедино тех шагов, которые мы уже прошли (или предполагается, что прошли) к этому моменту.

6.1. Соединение компонентов решения

На данный момент мы знаем:

- как выбрать и реализовать архитектуру приложения Kotlin Multiplatform;
- как работать с многопоточностью;
- что делать общим, а что нативным;
- как писать сетевой клиент.

Если вы читаете книгу последовательно, то, вероятно, у вас уже практически готово наше приложение. Если нет, то самое время напомнить, что нам нужно сделать.

Если вы открыли книгу на этой главе, то советую вернуться к главе 2 об архитектуре и каркасе нашего приложения, создать шаблон и пошагово заполнить его кодом сейчас по приведенному плану. Вы также можете ознакомиться с готовым листингом по ссылке в конце книги.

У нас уже готов сетевой слой, мы подробно разобрали его в предыдущей главе (глава 5). Также в главе о многопоточности (глава 4) мы добавили диспетчеры корутин в commonMain (листинг 6.1).

Листинг 6.1. Диспетчеры корутин

```
val uiDispatcher = Dispatchers.Main
val ioDispatcher = Dispatchers.IO
```

В качестве общего архитектурного компонента у нас используется написанная нами заготовка `ViewModel` (листинг 6.2).

Листинг 6.2. Начало NewsViewModel

```
class NewsViewModel {}
```

Чтобы вызывать из нее код бизнес-логики, добавим базовый класс для `ViewModel` со скоупом корутин (листинг 6.3).

Листинг 6.3. Базовая ViewModel со scope

```
open class BaseViewModel(
      protected val scope: CoroutineScope = CoroutineScope(uiDispatcher),
) {}
```

Теперь реализуем наследование `NewsViewModel` от базовой вьюмодели BaseViewModel (листинг 6.4).

Листинг 6.4. Донастройка NewsViewModel

```
class NewsViewModel() : BaseViewModel() {

  fun loadNews() {
    scope.launch {
      // Вызов из клиента
    }
  }
}
```

Мы можем, конечно, вызывать сетевой клиент напрямую из `ViewModel`, но правильнее будет инкапсулировать запросы, включая работу с многопоточностью. Для этого мы выполним следующие шаги:

1. Добавим специальный сервис `NewsService` для инкапсуляции запросов к сетевому клиенту.
2. Выделим запрос к сервису в `UseCase`. Этот механизм поможет нам не только сделать переиспользуемый код, но и добавить обработку ошибок, а также вызов в скоупе фоновой корутины.

Наш сервис будет иметь примерно такой код, как показано в листинге 6.5.

Листинг 6.5. Сервис новостей

```
class NewsService() {
        // TODO: переделать на DI
        private val httpClient: NetworkClient = NetworkClient()


    suspend fun loadNews(): Result<NewsList> {
        return httpClient.request(URL)
```

```
    }

    companion object {
        val  URL = "https://newsapi.org/v2/everything?q=science"
    }
}
```

Для тестирования на текущем этапе мы можем инициализировать сетевой клиент прямо в `NewsService`. Но постоянное использование приведет к неконсистентному состоянию. В следующей главе (глава 7) мы свяжем компоненты правильно с помощью DI.

Займемся `UseCase`. Сначала создадим базовый класс с оберткой для выполнения кода в скоупе корутин (листинг 6.6).

Листинг 6.6. Базовый UseCase

```
import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.withContext

abstract class BaseUseCase<in T, out R>(
        private val dispatcher: CoroutineDispatcher = ioDispatcher
) {

  abstract suspend fun execute(param: T): R

  suspend operator fun invoke(param: T): Result<R> = withContext(dispatcher) {
    runCatching { execute(param) }
  }
}
```

В нем мы определяем оператор invoke как `suspend`-функцию, которая нам вернет результат выполнения. Благодаря `runCatching` мы сможем отловить ошибку при выполнении метода execute и вернуть исключение как вариант результата.

Для гибкости реализуем базовый `UseCase` как generic.

Теперь добавим специальный `NewsUseCase`, в котором переопределим метод execute (листинг 6.7).

Листинг 6.7. Код NewsUseCase

```
class NewsUseCase (): BaseUseCase<Unit, NewsList?>() {

  // TODO: переделать на DI
  private val newsService: NewsService = NewsService()

  override suspend fun execute(param: Unit): NewsList? {
    return newsService.loadNews()
  }
}
```

Связь NewsUseCase с NewsService тоже надо будет переделать — этим мы займемся в следующей главе. Пока оставим ссылку на сервис в виде приватного свойства, которому присвоим значение, создаваемое прямо здесь.

В методе execute вызовем метод loadNews сервиса NewsService.

Соединим теперь NewsUseCase с нашей ViewModel (листинг 6.8).

Листинг 6.8. Код NewsViewModel

```
class NewsViewModel() : BaseViewModel() {
  // TODO: переделать на DI
  private val useCase: NewsUseCase = NewsUseCase()
  var newsFlow = MutableStateFlow<NewsList?>(null)

  fun loadNews() {
    scope.launch {
            // Запуск
      val result = useCase.invoke(Unit)
            // Обработка результатов
      result.getOrNull()?.let {
        newsFlow.tryEmit(it)
      }
    }
  }
}
```

Для получения данных будем использовать Kotlin StateFlow, на который подпишемся со стороны нативных клиентов. Для получения данных или ошибки мы просто вызываем метод invoke с передачей пустого блока. Затем обрабатываем результат.

Теперь нужно добавить подписку и получение данных из newsFlow на стороне iOS-приложения. Создадим класс-обертку для обозревателя Kotlin StateFlow и реализуем протокол коллектора потоков. Для простоты обращения создадим псевдоним типа через typeAlias (листинг 6.9).

Листинг 6.9. Обертка для подключения к Kotlin Flow со стороны iOS

```
import Foundation
import shared

typealias Collector = Kotlinx_coroutines_coreFlowCollector

class Observer: Collector {
  func emit(value: Any?, completionHandler: @escaping (Error?) -> Void) {
    callback(value)
    completionHandler(nil)
  }

  let callback:(Any?) -> Void
```

```
  init(callback: @escaping (Any?) -> Void) {
    self.callback = callback
  }
}
```

Для Android Kotlin `StateFlow` — это нативный механизм, поэтому дополнительно для нативного Android-приложения здесь ничего делать не надо.

Теперь подключим `Observer` к классу модели состояния (листинг 6.10).

Листинг 6.10. Подключение к NewsListVm Observer

```
class NewsListVm: ObservableObject {

  @Published var news: [NewsItem] = []
  // Коллектор Flow
  lazy var itemsCollector: Observer = {
    let collector = Observer { [weak self] value in
      guard let self = self else {return}
      if let value = value as? NewsList {
        // Присваивание результата
        self.news = value.articles ?? [NewsItem]()
      }
    }
    return collector
  }

  // VM из общего модуля
  private lazy var vm: NewsViewModel? = {
    // TODO: переделать на DI
    let vm = NewsViewModel()
    vm?.newsFlow.collect(
      collector: itemsCollector, completionHandler: {_ in})
    return vm
  }()

  func loadNews()  {
    // Загрузка
    vm?.loadNews()
  }
}
```

Выделим `itemsCollector`, внутри которого реализуем инстанс `Observer` с передачей блока завершения. В этом блоке обратного вызова добавим обработку полученных данных.

Соединим `itemsCollector` с `NewsViewModel`. Создадим lazy-инстанс `NewsViewModel` и подключим `itemsCollector` к методу collect `newsFlow` этой `ViewModel`. Теперь можем вызвать логику общей `NewsViewModel` в методе loadNews. Добавим вызов в View (листинг 6.11).

Листинг 6.11. Запуск из View на iOS

```
struct NewsListView: View {
  @ObservedObject var model = NewsListVm()

  var body: some View {
    NavigationView {
      List(model.items, id: \.self) { item in
        NavigationLink(destination: NewsItemView(item: item)) {
          NewsItemRow(item: item)
        }
      }.onAppear {
        // Загрузка новостей из ViewModel
        model.loadNews()
      }
    }
  }
}
```

На стороне Android добавим передачу `NewsViewModel` из commonMain и подписку на `StateFlow` прямо в `Composable` (листинг 6.12). Для этого таргета такое поведение абсолютно корректно.

Листинг 6.12. Подключение ViewModel к Composable NewsListScreen

```
@Composable
fun NewsListScreen(viewModel: NewsViewModel) {
  LaunchedEffect(Unit) {
   // Загрузка новостей
    viewModel.loadNews()
  }
  val newsList: NewsList? by viewModel.newsFlow.collectAsState()

  // Передача данных в Composable списка
  NewsListView(newsList?.articles.orEmpty(), onClick)
}
```

Добавим подписку к новостям через метод `collectAsState`.

Соединим `Composable` с `Activity`, как в листинге 6.13. Обратите внимание, что сейчас мы инициализируем модель в `Activity` и передаем в Composable, чтобы избежать пересоздания при перестройке функций UI.

Листинг 6.13. Передача ViewModel в Composable со стороны Activity

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContent {
    // Создаем ViewModel и передаем в Composable
    NewsListScreen(NewsViewModel())
  }
}
```

Проверим результат (рис. 6.1).

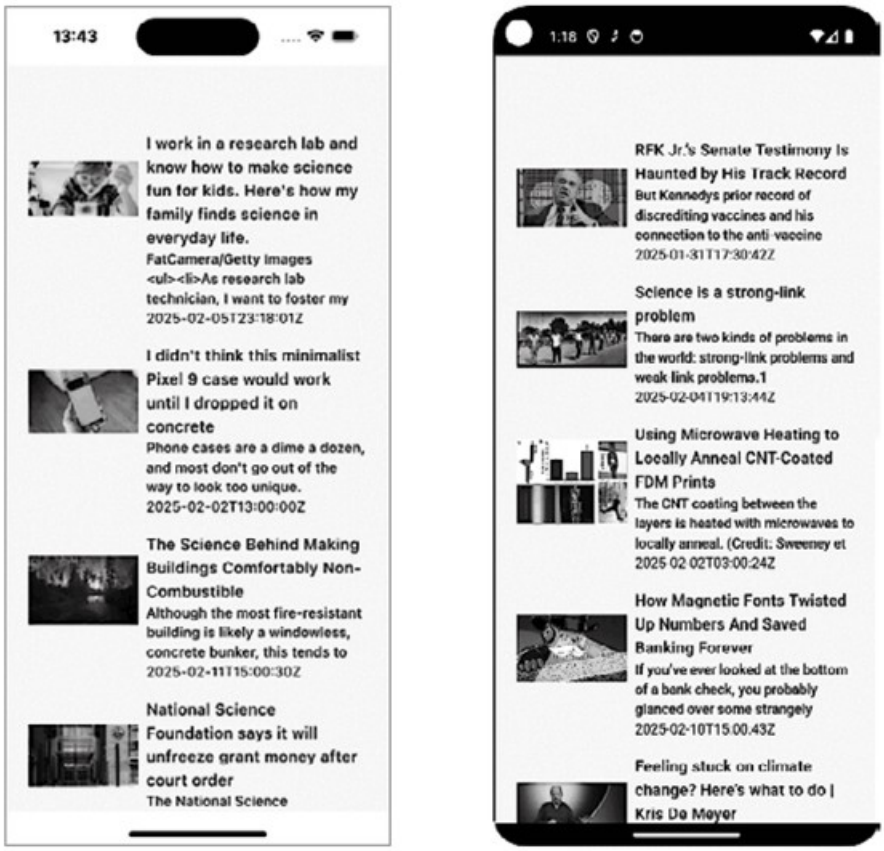


Рис. 6.1. Результат работы приложения

Итак, у нас есть приложение на KMP, но его можно улучшить. Давайте определим, что для этого необходимо сделать:

1. Соединить слои приложения с помощью Dependency Injection.
2. Добавить функцию добавления/удаления в избранное и хранения данных локально.
3. Перейти на общий UI.
4. Добавить еще один экран для отдельной новости и навигацию на него.

Этим мы займемся в следующих главах. А сейчас посмотрим, что из кода можно оптимизировать, заменив на готовое решение. Речь пойдет об использовании специальной кросс-платформенной версии `ViewModel` от Google.

6.2. Миграция на готовую ViewModel

Осенью 2022 года компания Google взяла курс на поддержку Jetpack Compose для Kotlin Multiplatform. Несмотря на то что Google разрабатывает другую

известную кросс-платформенную технологию, Flutter, компания решила поддержать и SDK KMP как перспективный и удобный инструмент. Также стоит учитывать, что Kotlin Multiplatform — это одна из технологий экосистемы Kotlin, официального языка разработки Android, и, как известно, компания придерживается подхода Kotlin-First.

С мая 2024 года Google официально поддерживает KMP как технологию, которая позволяет совместно использовать код на мобильных, веб-, серверных и десктопных платформах. Это помогает повысить производительность разработчиков, больше внимания уделять качеству и отлично вписывается в подход Android, основанный на использовании Kotlin.

Совместно с инженерами JetBrains команды инженеров Google активно занимаются развитием и разработкой Kotlin Multiplatform. В числе целей на будущее намечены:

- расширение и стабилизация библиотек Jetpack с поддержкой Kotlin Multiplatform;
- поддержка целевой платформы Wasm в библиотеках Jetpack;
- повышение производительности сборки на Kotlin/Native;
- отладка на Kotlin/Native;
- расширение поддержки KMP в Android Studio.

Подробнее со списком поддерживаемых решений для KMP можно ознакомиться здесь: https://developer.android.com/kotlin/multiplatform. Обратите внимание, что не все версии решений Jetpack поддерживают мультиплатформу.

Давайте попробуем `ViewModel` от Google и сравним ее работоспособность с тем, что реализовали мы.

Для начала добавим подходящую зависимость в `commonMain.dependencies` (листинг 6.14).

Листинг 6.14. Подключение зависимостей lifecycle-viewmodel

```
sourceSets {
    commonMain.dependencies {
        val lifecycle = "2.8.0-rc01"
        implementation("androidx.lifecycle:lifecycle-viewmodel:$lifecycle")
    }
}
```

Поддерживается только `Viewmodel`. Не добавляйте все, что входит в пакет `lifecycle`, иначе получите ошибки, как в листинге 6.15.

Листинг 6.15. Возможные ошибки при неверно подключенных зависимостях

```
Execution failed for task ':shared:transformIosMainCInteropDependencies
                            MetadataForIde'.
```

```
> Could not resolve all files for configuration ':shared:iosX64Compilation
                                                  DependenciesMetadata'.
> Could not resolve androidx.lifecycle:lifecycle-runtime-ktx:2.8.0-alpha03.
  Required by:
      project :shared
      project :shared > androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.0-alpha03
      project :shared > androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.0-
               alpha03 > androidx.lifecycle:lifecycle-viewmodel:2.8.0-alpha03 >
               androidx.lifecycle:lifecycle-viewmodel-iosx64:2.8.0-alpha03
> No matching variant of androidx.lifecycle:lifecycle-runtime-ktx:2.8.0-
alpha03 was found. The consumer was configured to find a library for use during
'kotlin-metadata', preferably optimized for non-jvm, as well as attribute
'org.jetbrains.kotlin.platform.type' with value 'native', attribute
'org.jetbrains.kotlin.native.target' with value 'ios_x64' but:
  - Variant 'androidxSourcesElements' capability androidx.lifecycle:lifecycle-
runtime-ktx:2.8.0-alpha03:
    - Incompatible because this component declares documentation for use during
'androidx-multiplatform-docs' and the consumer needed a library for use during
'kotlin-metadata'
    - Other compatible attributes:
      - Doesn't say anything about its target Java environment (preferred
optimized for non-jvm)
      - Doesn't say anything about org.jetbrains.kotlin.native.target
(required 'ios_x64')
      - Doesn't say anything about org.jetbrains.kotlin.platform.type
(required 'native')
```

Изменим нашу базовую `ViewModel` (листинг 6.16) так, чтобы в качестве базового класса она наследовала `ViewModel`. Также вместо создаваемого скоупа мы можем ограничиться `ViewModelScope`.

Листинг 6.16. Смена базовой ViewModel

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope

open class BaseViewModel : ViewModel() {
  val scope = this.viewModelScope
}
```

А в `NewsViewModel` либо оставим все по-прежнему, либо напрямую обратимся к родительскому скоупу (листинг 6.17).

Листинг 6.17. Смена scope на родительский в NewsViewModel

```
class NewsViewModel() : BaseViewModel() {
  private val useCase: NewsUseCase = NewsUseCase()
  var newsFlow = MutableStateFlow<NewsList?>(null)

  fun loadNews() {
    // используем scope из ViewModel()
    viewModelScope.launch {
      val result = useCase.invoke(Unit)
      result.getOrNull()?.let {
```

```
                newsFlow.tryEmit(it)
            }
        }
    }
}
```

Получим тот же результат, что и с нашей рукотворной `ViewModel`. Но решение от Google может оказаться быстрее и производительнее.

6.3. Материал со звездочкой. Что под капотом ViewModel

Как и в традиционной `ViewModel` для Android и JVM, нам доступен встроенный `ViewModelScope`. Его и будем использовать вместо собственного.

Посмотрим на внутреннее устройство его механизма. Фрагмент кода показан в листинге 6.18.

Листинг 6.18. Код ViewModelScope

```
/**
 * The [CoroutineScope] associated with this [ViewModel].
 *
 * The [CoroutineScope.coroutineContext] is configured with:
 * - [SupervisorJob]: ensures children jobs can fail independently of each
     other.
 * - [MainCoroutineDispatcher.immediate]: executes jobs immediately on the main
     (UI) thread. If
 *   the [Dispatchers.Main] is not available on the current platform (e.g.,
     Linux), we fallback
 *   to an [EmptyCoroutineContext].
 *
 * This scope is automatically cancelled when the [ViewModel] is cleared,
   and can be replaced by
 * using the [ViewModel] constructor overload that takes in a `viewModelScope:
   CoroutineScope`.
 *
 * For background execution, use [kotlinx.coroutines.withContext] to switch
   to appropriate
 * dispatchers (e.g., [kotlinx.coroutines.IO]).
 *
 * @see ViewModel.onCleared
 */
public val ViewModel.viewModelScope: CoroutineScope
        get() = synchronized(VIEW_MODEL_SCOPE_LOCK) {
            getCloseable(VIEW_MODEL_SCOPE_KEY)
                    ?: createViewModelScope().also { scope -> addCloseable
                       (VIEW_MODEL_SCOPE_KEY, scope) }
        }

private val VIEW_MODEL_SCOPE_LOCK = SynchronizedObject()
```

Внутри используется потокобезопасный лок как механизм синхронизации, реализованный на уже знакомом нам подходе `expect/actual`.

Функция `createViewModelScope()` внутри создает собственный скоуп корутин, куда подставляется `Dispatchers.Main` (листинг 6.19).

Листинг 6.19. Создание viewModelScope

```
internal fun createViewModelScope(): CloseableCoroutineScope {
  val dispatcher = try {
    Dispatchers.Main.immediate
  } catch (_: NotImplementedError) {
    // In platforms where `Dispatchers.Main` is not available, Kotlin
       Multiplatform will throw
    // a `NotImplementedError`. Since there's no direct functional alternative,
    // we use `EmptyCoroutineContext` to ensure a `launch` will run in the same
    // context as the caller.
    EmptyCoroutineContext
  }
  return CloseableCoroutineScope(coroutineContext = dispatcher +
         SupervisorJob())
}
```

> **ВНИМАНИЕ!** Если мы работаем с таргетом, который не имеет своей реализации `Dispatchers.Main`, например Linux, то получим исключение EmptyCoroutineContext. Следовательно, `ViewModelScope` мы использовать не сможем.

Еще одно новшество API `ViewModels` — возможность переопределять `viewModelScope` и передавать скоупы как параметр `ViewModel` (листинг 6.20).

Листинг 6.20. Пример кода для тестирования ViewModel

```
class MyViewModel(
        // Make Dispatchers.Main the default, rather
           than Dispatchers.Main.immediate
        viewModelScope: CoroutineScope = Dispatchers.Main + SupervisorJob()
) : ViewModel(viewModelScope) {
        // Use viewModelScope as before, without any code changes
}

// Allows overriding the viewModelScope in a test
fun Test() = runTest {
        val viewModel = MyViewModel(backgroundScope)
}
```

Для тестирования есть специальный backgroundScope, который рекомендовано использовать.

Что ж, если не учитывать проведенную работу по оптимизации диспетчеров корутин и инкапсуляции работы с многопоточностью, то может создаться впечатление, что разработчики просто взяли наш код и поместили его в общую библиотеку.

API библиотеки для кросс-платформы также включает в себя: `ViewModelStore`, `ViewModelStoreOwner` и `ViewModelProvider`. `ViewModelProvider` поддерживает теперь запрос инстансов по типу не только как `java.lang.Class`, но и как `kotlin.reflect.KClass`. `ViewModelProvider.NewInstanceFactory` и `ViewModelProvider.AndroidViewModelFactory` доступны только для Android и JVM, и их использование на других таргетах выдаст ошибку: `UnsupportedOperationException`.

Для всех не-JVM-таргетов теперь надо реализовывать собственные фабрики на основе `ViewModelProvider.Factory` с переопределением метода `create` (листинг 6.21).

Листинг 6.21. Создание ViewModelProvider.Factory

```
class CustomFactory: ViewModelProvider.Factory {

  override fun <T : ViewModel> create(modelClass: KClass<T>, extras:
CreationExtras): T {
    return super.create(modelClass, extras)
  }
}
```

Если мы будем получать инстанс `ViewModel` через DI, ничего не изменится.

Также с версии 2.9.0-alpha07 поддерживается сохранение состояния `ViewModel` для Kotlin Multiplatform. Это позволяет использовать `SavedStateHandle` в общем коде (листинг 6.22).

Листинг 6.22. Пример использования сохранения состояния ViewModel

```
@Serializable
data class Person(val firstName: String, val lastName: String)

class MyViewModel(handle: SavedStateHandle) : ViewModel() {
  var person by handle.saved { Person("John", "Doe") }

  fun onPersonChanged(person: Person) {
    this.person = person
  }
}
```

`SavedStateHandle` позволяет сохранять сериализуемые объекты. В наших практических примерах пока такой функционал не требовался, но он будет необходим, если у вас будут формы с вводом или разноуровневая навигация.

Также с версии 2.9.0-alpha01 для тестирования `ViewModel` в изоляции с помощью класса `ViewModelScenario` (https://developer.android.com/reference/kotlin/androidx/lifecycle/viewmodel/testing/ViewModelScenario) поддерживается библиотека `androidx.lifecycle:lifecycle-viewmodel-testing`. Теперь всем таргетам доступна очистка ресурсов `ViewModel` в `onCleared`. `ViewModel` и `ViewModelProvider` также стали потокобезопасны.

ВНИМАНИЕ! Несмотря на то что для версий 2.9.0 указана поддержка Kotlin Multiplatform, часть функционала в них поддерживает только Android, поэтому при установке в commonMain вы можете получить ошибку. Некоторые из этих версий уже недоступны для подгрузки, поэтому на данный момент проверенной является версия 2.8.0-rc01.

ПОДВЕДЕМ ИТОГИ

- ✓ У нас есть официальный API, который дает нативную реализацию `ViewModel`. Всю рутину теперь делают за нас. Но при необходимости мы по-прежнему можем переопределять скоупы на свой вкус.
- ✓ Миграция проходит максимально быстро и бесшовно.
- ✓ В следующей главе мы доработаем наше решение, добавив поддержку Dependency Injection.

ГЛАВА 7

Связываем слои. DI в Kotlin Multiplatform

В этой главе рассмотрим:

- вариации архитектуры и связей в приложении KMP;
- виды Dependency Injection;
- ручной DI для решения проблем с общим кодом;
- готовые решения для KMP;
- ограничения в Kotlin Native и способы решения;
- доработку нашего приложения, подключение и настройку DI.

7.1. Концепция общего Dependency Injection (DI)

Kotlin Multiplatform предоставляет механизмы для совместного использования большей части кода на разных платформах. Это всевозможные сервисы для работы с сетью и базами данных. Безусловно, это различные модели данных, тем более что языки Kotlin и Swift довольно похожи по синтаксису. Для создания единой структуры кросс-платформенного приложения рекомендуется делать также общей и архитектуру: общий интерактор, презентер, `ViewModel`. В нативных частях приложения у нас остается работа с собственно платформой и с различными датчиками, а также UI. На момент написания книги уже появились способы создания общих View на Compose (поговорим об этом в следующих главах), а также использования `ViewModel` не только как части паттерна MVVM, но и с помощью библиотеки Jetpack (этот способ рассмотрен в главе 6, где мы говорим об архитектуре).

Возможность реализовывать общий код по-разному доступна, как мы помним, благодаря механизму `expect/actual`. Нам ничто не мешает задекларировать в общем модуле только интерфейс какого-то компонента, например сервиса, и реализовать его по-разному в разных нативных приложениях, как показано на рис. 7.1.

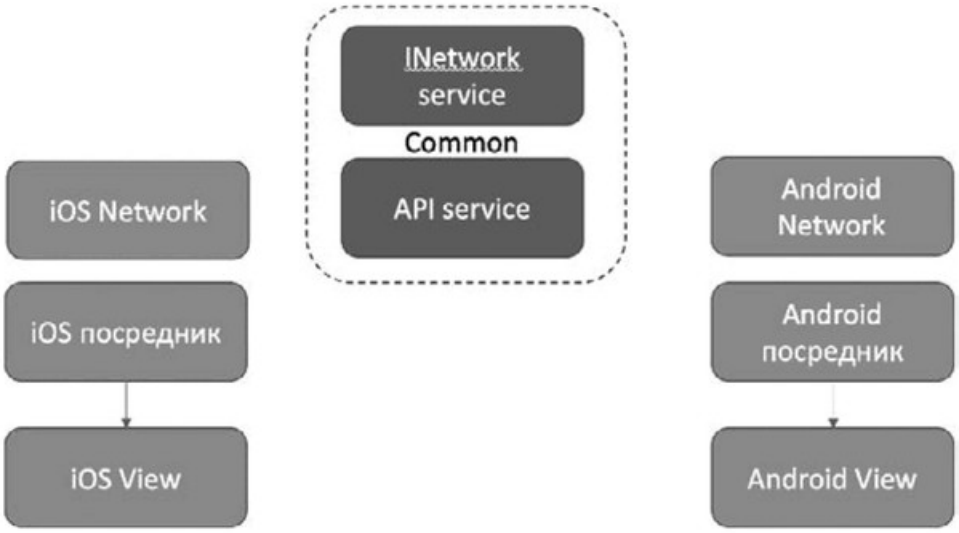


Рис. 7.1. Схема облегченного приложения

Это возможный подход, но в таком случае у нас получается набор разрозненных компонентов или сервисов, которые мы можем вызывать по отдельности из нативной части, и это в целом нарушает идею кросс-платформенного приложения. Также могут дублироваться инстансы объектов, возникает неконсистентность данных. В целом это не так эффективно, как использовать полностью реализацию в shared-модуле.

Мы вполне можем использовать общий модуль KMP как набор отдельных сервисов и собирать всю структуру на стороне нативного приложения (рис. 7.2).

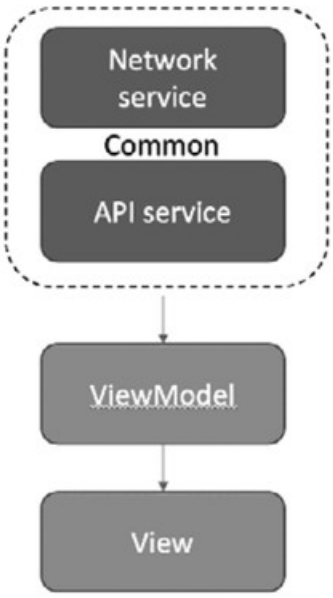


Рис. 7.2. Связь слоев в одном таргете

Какой бы подход мы ни выбрали, у нас возникает вопрос, как правильно связывать слои логики в приложении. В shared-модуле у нас общая бизнес-логика и слой архитектуры, в нативных приложениях — View (рис. 7.3).

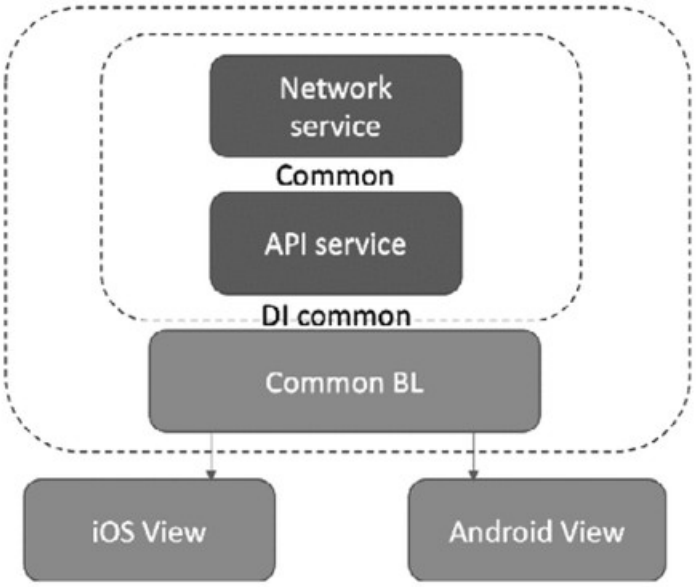


Рис. 7.3. Приложение с общей бизнес-логикой и архитектурой

Для того чтобы корректно связать друг с другом слои бизнес-логики с архитектурными компонентами и View, нам потребуется использовать механизм Dependency Injection прежде всего в нашей common-части. Это и является рекомендованным способом решения задачи построения структуры приложения, в том числе и от самой компании JetBrains.

Для начала — пара слов о том, что такое Dependency Injection. Не путайте с Dependency Inversion из принципов SOLID, хотя эти понятия и связаны. Инверсия зависимостей — это принцип зависимости слоев в иерархии, когда слои верхнего уровня не зависят от слоев нижнего. Принцип Dependency Injection заключается в предоставлении зависимостей одного компонента от другого через так называемую инжекцию. Передачу зависимостей можно реализовывать как через конструктор-инициализатор, так и через свойства (листинг 7.1).

Листинг 7.1. Пример передачи зависимостей в сущности

```
class NewsService(private val httpClient: NetworkClient) {}
class NewsUseCase (private val newsService: NewsService){}
```

Для настройки и создания компонентов, а также отслеживания корректного предоставления зависимостей в реализации паттерна DI используются контейнер и сервис-локатор (ServiceLocator). Назначение сервис-локатора — хранить существующие инстансы и предоставлять их по запросу. Реже — создавать новые инстансы (листинг 7.2).

Листинг 7.2. Пример класса сервис-локатор object ServiceLocator {

```
  fun <reified T> register(factory: () -> T)
  fun <reified T> resolve(): T
}

class NewsService {
  private val network: NetworkService = ServiceLocator.resolve()
}

fun main() {
  ServiceLocator.register<NetworkService> { NetworkService() }
  val newsService = NewsService()
}
```

Обычно реализуется два generic-метода: один — для подключения зависимости, другой — для ее регистрации. Затем уже в необходимой точке нашего приложения мы сначала сохраняем все зависимости, а потом уже можем вызывать их (листинг 7.3).

Листинг 7.3. Реализация generic сервис-локатора

```
object ServiceLocator {
  var services: MutableMap<String, Any> = mutableMapOf()

  inline fun <reified T: Any> add(service: T) {
    (T::class.qualifiedName)?.let {
      services.put(it, service)
    }
  }

  inline fun <reified T: Any> service(): T? {
    return ((T::class.qualifiedName)?.let {
      services[it]
    } as? T) ?: null
  }
}
```

DI-контейнер, наоборот, используется больше для создания инстансов, чем для их хранения. Реализовывать его можно с использованием фабрики типа создаваемого инстанса, как показано в листинге 7.4.

Листинг 7.4. Пример DI-контейнера

```
class DIContainer {
    private val instances = mutableMapOf<Class<*>, Any>()

    fun <T:Any> register(clazz: Class<T>, instance: T) {
        instances[clazz] = instance
    }

    fun <T:Any> resolve(clazz: Class<T>): T {
          // Получаем инстанс по типу или создаем его
```

```
        return instances[clazz] as? T?: createInstance(clazz)
    }

    private fun <T:Any> createInstance(clazz: Class<T>): T {
        val constructor = clazz.declaredConstructors.first()
        val parameters = constructor.parameterTypes.map {
resolve(it) }.toTypedArray()

           //Вызов фабрики
        return constructor.newInstance(*parameters) as T
    }
}
```

В целом это все, что нам необходимо для самостоятельной реализации ручного DI. В небольших приложениях обычно и довольствуются таким минималистичным решением (листинг 7.5).

Листинг 7.5. Пример упрощенного DI для небольшого приложения

```
object DI {
  val httpClient: NetworkClient by lazy {
    NetworkClient()
  }

  val newsService: NewsService by lazy {
    NewsService(httpClient)
  }
}
```

Здесь мы просто создаем синглтон, а каждый нужный нам инстанс получаем из lazy-свойства.

Однако при реализации уже большого серьезного приложения мы должны задумываться о таких вещах, как:

- потокобезопасность (доступ к инстансам из разных потоков);
- управление памятью и поддержка слабых ссылок для предотвращения утечек памяти;
- управление количеством инстансов объекта в разных скоупах, в том числе и временем жизни ссылки или объекта;
- создание и использование синглтонов.

Если речь идет о кросс-платформенной разработке, то по всем этим пунктам возникают определенные нюансы, поскольку на iOS и Android они решаются по-разному из-за особенностей языков и SDK. Эти различия необходимо учитывать при разработке общего решения. В настоящий момент часть вопросов уже решена за нас, и нам достаточно воспользоваться готовым «нативным» кросс-платформенным инструментарием, например, для реализации потокобезопасности и управления памятью. Но то, насколько правильно они будут применяться, зависит уже от нас.

Итак, наша цель — хорошо и правильно работающий DI для обеих платформ. Решение, в котором учитываются нюансы только одного таргета, нам не подойдет.

Давайте подумаем, что должно быть в хорошем Dependency Injection-решении. Все то, что стандартно для работающего решения: работоспособность, устойчивость к ошибкам и возможность отладки. В расширенной версии потребуется реализовать и отслеживание циклических зависимостей circle dependencies. Для правильной реализации DI в Kotlin Multiplatform (а также на любой другой кросс-платформе) будут важны следующие критерии:

1) управление потоками;
2) управление памятью (weak);
3) инициализация (Kotlin Objective-C interop);
4) соотношение типов (Kotlin Objective-C interop);
5) синглтоны.

Управление потоками и управление памятью сейчас реализуются с помощью инструментария Kotlin Native. Мы подробно рассмотрели это в главах 3 и 4 (общее/нативное и многопоточность). Поэтому сфокусируемся на пунктах 3, 4 и 5.

Прежде чем погружаться в академические дебри создания своего DI, давайте посмотрим, можем ли мы использовать готовые решения.

При работе с нативным Android-приложением стандартным является использование таких библиотек, как Dagger или Hilt. В текущей реализации для common-части нашего KMP-проекта они не подойдут, потому что явно применяют механизм JVM. Например, в качестве служебных маркеров применяются аннотации, обработка которых сейчас идет все еще с помощью KAPT (Kotlin Annotation Processing Tool), который не поддерживается Kotlin Native. В настоящий момент ведутся работы по переводу библиотек на технологии, поддерживающие корректную работу не только под JVM (например, KSP — Kotlin Symbol Processing), но до полноценного релиза еще далеко. Вернее будет сказать так: KSP в Dagger уже внедрен, но сам Dagger пока работает только для Android.

Теперь посмотрим, что происходит на стороне iOS. Если не рассматривать сразу создание своего ручного DI, то у нас есть альтернатива в качестве готовой библиотеки Swinject, очень популярной среди разработчиков iOS. К сожалению, несмотря на всю красоту решения, Swinject есть только на стороне iOS. Swift-код мы в common-часть затащить не можем.

Можно выбрать подход, при котором используется свой нативный DI на стороне Android и свой механизм на стороне iOS. Но и в этом случае нам потребуется некое общее решение в common-части, как показано на рис. 7.4.

И в любом случае нам требуется решение и в common-части, потому что нам нужно соединять слои общей бизнес-логики между собой.

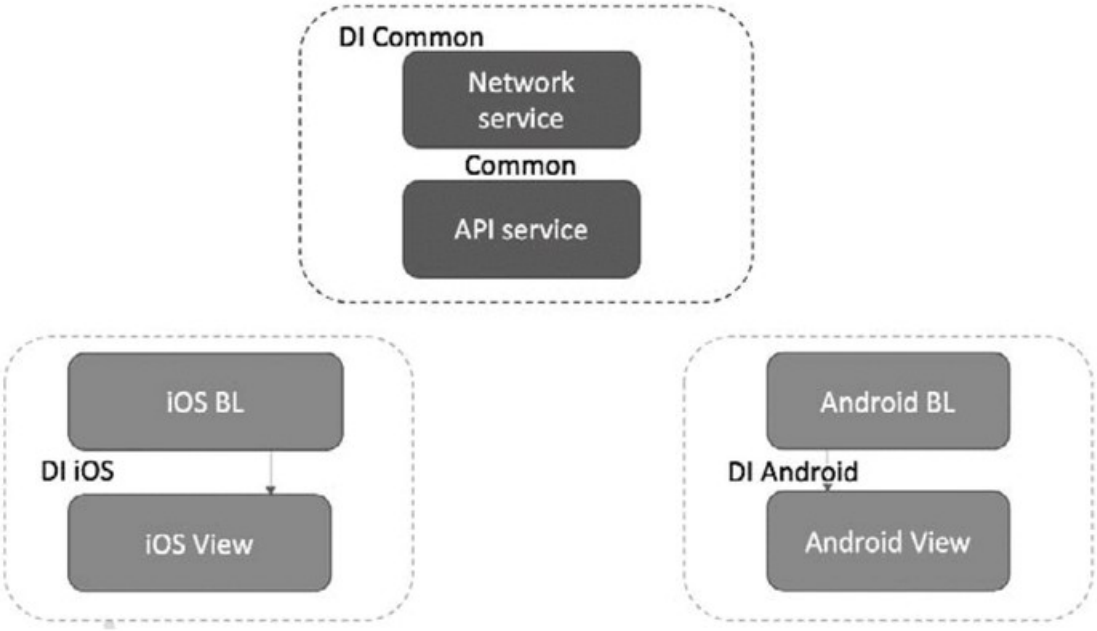


Рис. 7.4. Варианты использования DI

Можно было бы пофантазировать о том, как соединять нативный DI в общей части для единого управления с помощью подключения разных типов через typealias, но эта затея спорная сама по себе (листинг 7.6).

Листинг 7.6. Соединение разных типов DIManager через typealias

```
expect class DIManager

// iOS
actual typealias DIManager = SwinjectManager

// Android
actual typealias DIManager = HiltManager
```

Разумеется, в этом случае пришлось бы еще и определять общий интерфейс/протокол, чтобы иметь общие точки вызова бизнес-логики этих DIManager.

На наше счастье, Kotlin Multiplatform развивается, а энтузиасты и популяризаторы технологии создают новые решения, на которых мы можем протянуть до появления кросс-платформенного Dagger. Если же вы хотите узнать, как написать свой DI, то переходите по ссылке в конце книги. Это путь настоящих самураев.

В настоящий момент самыми популярными готовыми решениями для построения DI и архитектуры приложений KMP являются:

- Koin;
- Kodein;
- Kotlin Inject.

7.2. Начало работы с Kodein

Документацию по проекту вы можете найти на сайте https://kosi-libs.org/kodein/7.25/index.html, а саму библиотеку взять на github (https://github.com/kosi-libs/Kodein).

Это решение поддерживает и Kotlin Native, поэтому подходит и для нативного Android, и для кросс-платформы KMP. Скоупы и контексты Kodein поддерживает, но пока это работает только для Android.

Для подключения пропишем зависимость в `build.gradle` shared-модуля (листинг 7.7).

Листинг 7.7. Подключение Kodein в build.gradle.kts

```
val commonMain by getting {
  dependencies {
    implementation("org.kodein.di:kodein-di:*")
  }
}
```

Для регистрации компонентов Kodein использует DSL-синтаксис (листинг 7.8).

Листинг 7.8. Пример регистрации зависимостей в Kodein DI

```
val di = DI {
  bind<NewsUseCase> { provider { NewsUseCase(instance()) } }
  bind<NewsService> { singleton { NewsService(instance()) } }
  bind<NetworkClient> { singleton { NetworkClient() } }
}
```

Регистрация компонентов осуществляется с помощью команды `bind` внутри блока `DIAware.DI` с указанием типа. Также Kodein поддерживает работу и с синглтонами. Создание или получение инстансов классов реализуется одним из двух способов:

1. Чтобы у компонента (например, сервиса) был только один инстанс, используем блок `singleton`.
2. Если необходимо, чтобы инстанс создавался каждый раз при обращении, используем блок `provider` либо `factory` (листинг 7.9).

Листинг 7.9. Регистрация UseCase для создания инстанса с помощью Provider и Factory

```
// Provider и Factory создают инстанс каждый раз
bind<NewsUseCase> {
  provider { NewsUseCase(instance()) }
}

bind<NewsUseCase> {
  factory { service: NewsService ->
    NewsUseCase(service)
  }
}
```

Регистрация с помощью Provider подойдет, если нам не нужно передавать параметры для инициализации. Для передачи в инициализатор в качестве параметра компонента, инстансами которого также управляет DI, можно использовать instance (листинг 7.10).

Листинг 7.10. Пример кода расширения instance из исходного кода Kodein

```
public inline fun <reified T : Any> DirectDIAware.instance(tag: Any? = null): T =
        directDI.Instance<T>(generic(), tag)
```

Можно также использовать factory, чтобы автоматизировать инжекцию параметров.

Если у нас небольшое приложение, достаточно прямой регистрации в `DIAware.DI` через `bind`.

При переходе на сложную архитектуру, где для каждого экрана свой модуль или где мы хотим изолировать создание инстансов для сервисов и хранилищ в бизнес-логике, нам потребуется использовать специальные `DI.Module` (листинг 7.11).

Листинг 7.11. Вынос регистрации типов в модуль service

```
// Вынос регистрации типов в модуль
val serviceModule = org.kodein.di.DI.Module("service") {
  bindSingleton {
    NetworkClient()
  }
  bindSingleton {
    NewsService(instance())
  }
}
```

Подключить сами модули в `DIAware.DI` можно с помощью команды `import`, как в примере 7.12.

Листинг 7.12. Подключение модулей DIAware.DI

```
val di = org.kodein.di.DI {
  import(serviceModule)
  import(useCases)
  import(viewModels)
}
```

Kodein DI также подходит для работы с `ViewModel` и другими архитектурными компонентами.

После того как мы зарегистрировали все нужные нам сервисы и компоненты, нужно настроить их подключение. Kodein поддерживает lazy-инициализацию и работу с lazy-делегатами. На стороне Android-приложения мы можем вызвать подключение `ViewModel` через lazy-делегат, как показано в листинге 7.13.

Листинг 7.13. Подключение через lazy-делегат

```
class NewsActivity : AppCompatActivity {

    // Подключение NewsViewModel как инстанса
    val vm by di.instance<NewsViewModel>()
}
```

Для этого придется подключить зависимость Kodein для таргета Android.

iOS не поддерживает lazy-делегаты даже через интероп. Можно попробовать получить доступ напрямую через DirectDIAware в commonMain (листинг 7.14).

Листинг 7.14. Способы получения инстанса DI

```
object DIFactory {

  //Lazy
  val di = KodeinDI().di

  //Direct
  val diDirect = di.direct
}
```

Для унифицированного доступа к механизму Dependency Injection можно использовать фабрику-синглтон, которую разместим, разумеется, в common-части. Также для более удобного получения инстанса компонента по типу создадим generic-метод (листинг 7.15).

Листинг 7.15. Пример фабрики для Kodein DI

```
object DIFactory {
  val di by lazy {
    KodeinDI().di
  }
  val diDirect = di.direct

  // Запрос инстанса типа через generic
  inline fun<reified T> resolve():T? {
    return diDirect.instance()
  }
}
```

На стороне Android все работает красиво и элегантно (листинг 7.16).

Листинг 7.16. Запрос инстанса на стороне Android через resolve

```
class NewsActivity : AppCompatActivity() {

  //Вызываем generic-метод для получения инстанса
  val viewModel: NewsViewModel? = DIFactory.resolve()
}
```

А для IOS мы не учли, что inline-reified generics не поддерживаются при интеропе Kotlin в Swift: https://kotlinlang.org/docs/native-objc-interop.html#unsupported.

Этот механизм обеспечивает встраивание кода класса (кода реализации) в место вызова. Это очень специфический функционал сугубо для Kotlin JVM. В Swift поддерживается встраивание с помощью `@inlinable`, но сами inline-классы не поддерживаются. Мы разобрали этот момент в главе 3 про общее и нативное.

К сожалению, альтернативного метода для запроса по типу в Kodein нет. Мы не можем используемые типы заменить на Any (листинг 7.17).

Листинг 7.17. Код generic-метода для получения инстанса по типу в исходниках Kodein

```
* Gets an instance of `T` for the given type and tag.
 *
 * T generics will be erased.
 *
 * @param T The type of object to retrieve.
 * @param tag The bound tag, if any.
 * @return An instance.
 * @throws DI.NotFoundException if no provider was found.
 * @throws DI.DependencyLoopException If the instance construction triggered
   a dependency loop.
 */
public inline fun <reified T : Any> DirectDIAware.instance(tag: Any? = null): T {
  return directDI.Instance<T>(generic(), tag)
}
```

Для обхода ограничения можно попробовать создать свои внутренние фабрики для компонентов или конфигураторы, как в листинге 7.18.

Листинг 7.18. Возможная реализация фабрики ViewModel

```
object VMFactory {

    // Получение VM по типу экрана
    fun createVM(type: BaseView): ViewModel? {
        return when (type) {
            // Тип вычисляется явно
            is NewsView -> DIFactory.diDirect.instance<NewsViewModel>()
            else -> null
        }
        return null
    }

    // Generic для получения VM
    inline fun <reified T : ViewModel> getVM(): T? {
        return DIFactory.diDirect.instance() as? T
    }
}
```

В качестве типа-ключа можно использовать протокол. Например, в данном случае протоколы-наследники `BaseView` используются для получения `ViewModel` по типу.

Промежуточные выводы

Подведем итоги по работе с Kodein и проверим решение по нашему чек-листу. Данный DI подойдет либо для работы с Android, либо для работы только внутри common-модуля. Но для работы с iOS надо придумывать обходные решения. Проверяем по чек-листу:

1) инициализация (Kotlin Objective-C interop) +/-;
2) соотношение типов (Kotlin Objective-C interop) +/-;
3) синглтоны +.

Теперь давайте посмотрим, можно ли решить ту же самую задачу с помощью Koin.

7.3. Работа с Koin DI

Документация по проекту выложена на сайте (https://insert-koin.io/docs/setup/koin/), саму библиотеку можно найти на github (https://github.com/InsertKoinIO/koin).

Для установки добавим в `build.gradle` shared-модуля зависимость для commonMain. Для работы из общей логики нам потребуется Koin-Core (листинг 7.19).

Листинг 7.19. Подключение Koin Core

```
val commonMain by getting {
  implementation( "io.insert-koin:koin-core:*")
}
```

Текущие версии Koin рассчитаны на самые разные таргеты, включая не только Android или JS, но и WASM. Поддержка Kotlin Native включена в версию Core. Контексты и скоупы, так же как и в Kodein, поддерживаются только под Android-таргет. Кроме того, разработчики Koin подготовили решения с кодогенерацией на KSP (затронем эту тему в главе 11), а также плагин для Android Studio для упрощения работы.

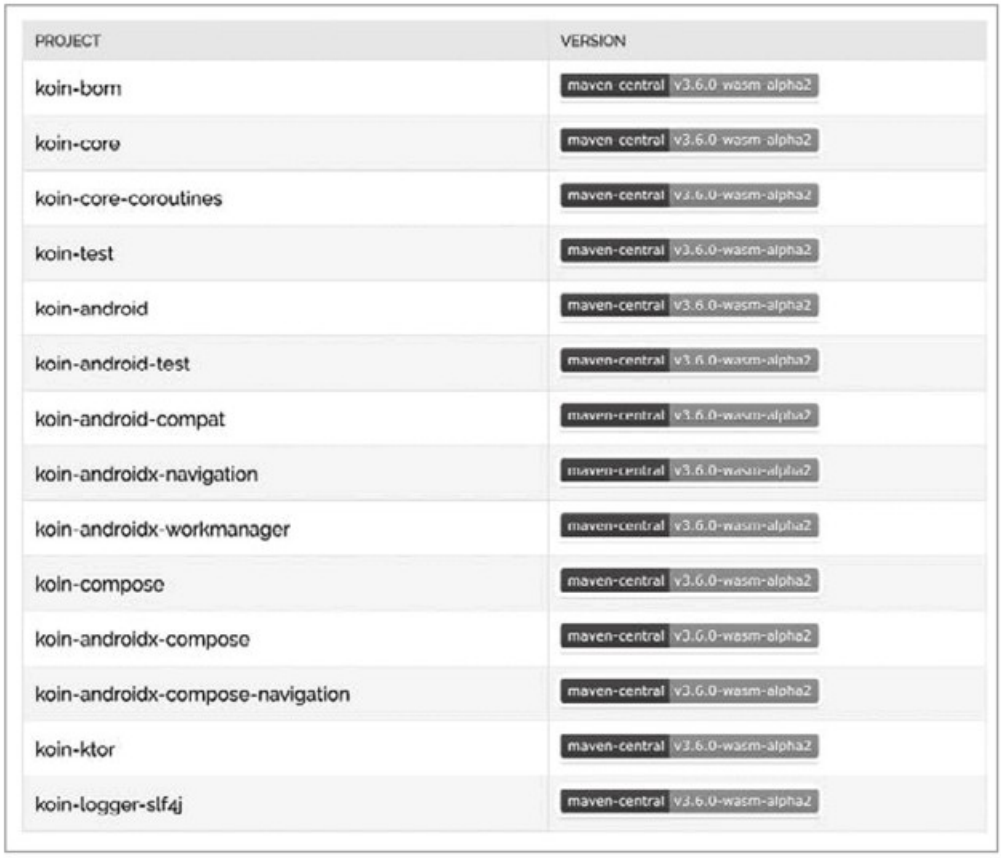
Не забывайте нажимать на «слона» для синхронизации зависимостей Gradle.

Для регистрации компонентов и зависимостей Koin, так же как и Kodein, использует DSL-синтаксис. Нам потребуется создать класс-наследник специального KoinComponent (листинг 7.20).

Листинг 7.20. Создание класса-наследника KoinComponent

```
class KoinDI : KoinComponent {

  // Код реализации
}
```



PROJECT	VERSION
koin-bom	maven central v3.6.0-wasm-alpha2
koin-core	maven central v3.6.0-wasm-alpha2
koin-core-coroutines	maven central v3.6.0-wasm-alpha2
koin-test	maven central v3.6.0-wasm-alpha2
koin-android	maven central v3.6.0-wasm-alpha2
koin-android-test	maven central v3.6.0-wasm-alpha2
koin-android-compat	maven central v3.6.0-wasm-alpha2
koin-androidx-navigation	maven central v3.6.0-wasm-alpha2
koin-androidx-workmanager	maven central v3.6.0-wasm-alpha2
koin-compose	maven central v3.6.0-wasm-alpha2
koin-androidx-compose	maven central v3.6.0-wasm-alpha2
koin-androidx-compose-navigation	maven central v3.6.0-wasm-alpha2
koin-ktor	maven central v3.6.0-wasm-alpha2
koin-logger-slf4j	maven central v3.6.0-wasm-alpha2

Рис. 7.5. Библиотеки Koin

Типы и инстансы регистрируются сразу в специальных модулях (листинг 7.21). Разумеется, можно использовать единый модуль, но лучше все-таки разделять по слоям логики и семантике.

Листинг 7.21. Регистрация сервисов в модуле

```
val serviceModule = module {
  single { NetworkClient() }
  single { NewsService(get()) }
}
```

Чтобы получить новый или существующий инстанс из зарегистрированных в `KoinComponent`, надо воспользоваться либо lazy-делегатом `inject()`, либо методом `get()` (листинг 7.22).

Листинг 7.22. Способы получения инстанса с lazy-делегатом и напрямую

```
class KoinDI : KoinComponent {

  // Получение через lazy
  val service: NewsService by inject()

  // Получение напрямую
  val service2: NewsService = get()
}
```

Koin также позволяет указать тип хранения и получения инстанса при регистрации (листинг 7.23).

Листинг 7.23. Регистрация типов как single или factory

```
val serviceModule = module {
    // Регистрируем как синглтон
    single { NetworkClient() }
    single { NewsService(get()) }
}

val usecaseModule = module {
    // Получаем инстанс при каждом обращении
    factory {
        NewsUseCase(get())
    }
}
```

Single позволяет зарегистрировать тип синглтоном. Если нам нужно получать новый инстанс каждый раз при обращении, то тип регистрируется через factory.

Регистрация модулей осуществляется через метод `startKoin`, результатом работы которого будет `KoinApplication`, это будет наша входная точка для обращения к DI, как показано в листинге 7.24.

Листинг 7.24. Создание KoinApplication через startKoin

```
fun start() = startKoin {
  modules(listOf(serviceModule, usecaseModule, vmModule))
}
```

Для обращения к DI создадим в `Common` синглтон со свойством `di`, к которому применим метод `start()` (листинг 7.25).

Листинг 7.25. Реализация фабрики для работы с DI

```
object KoinDIFactory {

  //Вызываем метод start при инициализации синглтона фабрики
  val di by lazy {
    KoinDI().apply {
      start()
    }
  }
}
```

Koin поддерживает запрос инстанса по типу через KClass, что позволит создать элегантное решение (листинг 7.26) в виде расширения, в котором тип будет передаваться в специальный метод get.

Листинг 7.26. Расширение для получения инстанса по типу как KClass

```
fun<T:Any> KoinDIFactory.resolve(clazz: KClass<*>):T? {
    return di.getKoin().get(clazz)
}
```

Напрямую с iOS использовать такой метод resolve для получения инстанса требуемого класса по-прежнему нельзя. В главе 3 про общее и нативное мы уже упоминали, что при конвертации типов между Objective-C/Swift и Kotlin возникают определенные нюансы, но в данном случае есть способ обойти эти ограничения.

Напомним, что Kotlin Native работает с `ObjCClass` вместо KClass (листинг 7.27). Подробно этот момент мы также разбирали в главе 3.

Листинг 7.27. Пример исходного когда из kotlinx.cinterop

```
  package kotlinx.cinterop
  import kotlin.native.*
  import kotlin.native.internal.*

  interface ObjCObject
  interface ObjCClass : ObjCObject
  interface ObjCClassOf<T : ObjCObject> : ObjCClass
// TODO: T should be added to ObjCClass and all meta-classes instead.
  typealias ObjCObjectMeta = ObjCClass

  interface ObjCProtocol : ObjCObject
```

Это интероп типов из iOS SDK в Kotlin. Чтобы не было путаницы между типами shared-модуля и других компонентов, к каждому типу со стороны commonMain общего модуля автоматически добавляется название модуля в виде префикса (листинг 7.28).

Листинг 7.28. Фрагмент кода из сгенерированного Objective-C shared-модуля

```
@interface SharedKoinDIFactory (Extensions)
- (id<SharedKotlinKClass> _Nullable)createTypeClazz:(Class)clazz __attribute__
((swift_name("createType(clazz:)")));
- (id _Nullable)resolveClazz:(id<SharedKotlinKClass>)clazz __attribute__((swift_
name("resolve(clazz:)")));
- (id _Nullable)resolveClazz_:(Class)clazz __attribute__((swift_
name("resolve(clazz_:)")));
@end
```

Хотя при запросе с iOS мы будем использовать типы из common-части, интероп сейчас сделает прямой запрос невалидным (листинг 7.29). Это равносильно передаче неизвестного незарегистрированного типа.

Листинг 7.29. Пример запроса по типу при передаче неизвестного типа

```
class NewsListModel : ObservableObject {
  private lazy var vm: NewsViewModel? = {
    let vm = KoinDIFactory.shared.resolve(
      clazz: NewsViewModel.self) as? NewsViewModel // Здесь будет null!!
    return vm
  }()
}
```

В итоге запрошенный инстанс будет равен `null`. Однако здесь мы можем получить KClass из `ObjCClass`. Добавим метод расширения нашей фабрики в iOSMain (листинг 7.30), в котором вызовем существующий метод преобразования `ObjCClass` в `KClass` `getOriginalKotlinClass`.

Листинг 7.30. Получение KClass через ObjCClass

```
fun<T:Any> KoinDIFactory.createType(clazz: ObjCClass):KClass<*>? {
  return getOriginalKotlinClass(clazz)
}
```

В принципе, мы могли бы воссоздать тип `ObjCClass` по его имени для более элегантного синтаксиса, как показано в листинге 7.31. Но из-за разницы в именах при интеропе это будет также невалидно.

Листинг 7.31. Попытка получить тип KClass из имени ObjCClass

```
fun<T:Any> createType(clazz: String):KClass<*>? {
    // По имени не найдет
    val objCClass = NSClassFromString(clazz)
    if (objCClass != null) {
        return getOriginalKotlinClass(objCClass)
    }
    return null // no type found
}
```

Этот вариант не работает. Остановимся на передаче `ObjCClass` напрямую и его конвертации в `KClass`.

Мы теперь знаем, как конвертировать один тип в другой. Добавим еще один метод расширения в iOSMain для получения инстанса из DI Factory по типу уже с учетом разницы платформ (листинг 7.32).

Листинг 7.32. Расширение для получения инстанса типа из DI

```
fun<T:Any> KoinDIFactory.resolve(clazz: ObjCClass):T? {
  val kClass = createType<T>(clazz)
  if (kClass != null) {
    return di.getKoin().get(kClass)
  }
  return null
}
```

Конфликта между iOSMain и commonMain не будет. Теперь новый метод можно безопасно вызывать со стороны iOS (листинг 7.33), так как код из iOSMain будет затрагивать только сам таргет и нативное приложение.

Листинг 7.33. Пример вызова кода со стороны iOS-приложения

```
class NewsListModel : ObservableObject {
  private lazy var vm: NewsViewModel? = {
    let vm = KoinDIFactory.shared.resolve(
      clazz: NewsViewModel.self) as? NewsViewModel // Теперь тут instance!!
```

```
    return vm
  }()
}
```

7.4. Синтаксические приемы для оптимизации кода

Можно добавить еще синтаксической красоты на стороне iOS-приложения. Для этого воспользуемся специальными структурами PropertyWrapper. Они используются для создания переменных состояния и автоматических свойств (листинг 7.34).

Листинг 7.34. PropertyWrapper для инжекции на стороне iOS

```
@propertyWrapper
struct Injected<T:AnyObject> {
    private var service: T?
    public  var container = DIFactory.shared

    public init(){}

    public var wrappedValue: T? {
        mutating get {
            if self.service == nil {
                self.service = container.resolve(type: T.self) as? T
            }
            return service
        }
        mutating set { service = newValue  }
    }
    public var projectedValue: Injected<T> {
        get { return self }
        mutating set { self = newValue }
    }
```

Нам потребуется создать обычную структуру-generic и пометить ее `@PropertyWrapper`. Container — это наша DI-фабрика из common-части. Переменная service используется для хранения ссылки на компонент, который мы будем запрашивать по типу.

Основное внимание нужно уделить свойству `wrappedValue`: если значение service у нас пустое, то надо запросить инстанс компонента из DI-фабрики.

Использовать такое свойство-обертку мы сможем, как показано в листинге 7.35. Обратите внимание, в данном коде используется макрос `@Observation`. Он упрощает управление моделью состояния.

Листинг 7.35. Пример использования @Injected

```
@Observation class NewsListModel {

    @Injected var service: NewsService
}
```

Хотя на Android проблем практически нет, мы и тут можем кое-что оптимизировать. Давайте создадим для нативного Android-приложения делегированное свойство (листинг 7.36).

Листинг 7.36. Реализация делегированного свойства для упрощенного подключения ViewModel

```
class ViewModels<T: Any?>(private val clazz: KClass<*>):
        ReadOnlyProperty<T, Any?> {

  private var weakReference:WeakReference<T>? = null

  override fun getValue(thisRef: T, property: KProperty<*>): Any? {
    if (weakReference == null) {
      weakReference = WeakReference(KoinDIFactory.resolve(clazz = clazz))
    }
    return weakReference?.get()
  }
}

inline fun<reified T:Any> viewModels() = ViewModels(T::class)
```

Для этого создадим свое свойство `ReadOnlyProperty`, в методе `getValue` которого обратимся к нашей DI-фабрике. Для управления памятью завернем отдаваемый инстанс в слабую ссылку. Затем создадим generic-метод для lazy-получения данных через наше свойство.

Также Koin поддерживает использование аннотаций для создания DI в стиле Dagger (пример в листинге 7.37).

Листинг 7.37. Подключение Koin с помощью аннотаций

```
@Module
class MyModule {

  @Single
  fun myComponent(myDependency: MyDependency) = MyComponent(myDependency)
}
```

Промежуточные выводы

Подведем итоги и проверим Koin по нашему чек-листу. Koin представляет собой более гибкое решение для работы с кросс-платформой с поддержкой типов `ObjCClass`:

1) инициализация (Kotlin Objective-C interop) +;
2) соотношение типов (Kotlin Objective-C interop) +;
3) синглтоны +.

Именно это решение мы и будем использовать как библиотеку DI для учебного приложения.

7.5. Kotlin Inject

Еще одним популярным DI можно назвать `Kotlin Inject`. Эта библиотека позволяет организовать и настроить связи между компонентами и работать с инстансами по типу `Dagger` (листинг 7.38).

Листинг 7.38. Пример настройки компонента через Kotlin Inject

```
@Component
abstract class AppComponent {
  abstract val repo: Repository

  @Provides
  protected fun jsonParser(): JsonParser = JsonParser()

  protected val RealHttp.bind: Http
    @Provides get() = this
}

interface Http

@Inject
class RealHttp : Http

@Inject
class Api(private val http: Http, private val jsonParser: JsonParser)

@Inject
class Repository(private val api: Api)
```

`Kotlin Inject` позволяет настроить работу со скоупами и компонентами (модулями в понимании Dagger), как в примере в листинге 7.39.

Листинг 7.39. Настройка scope в Kotlin Inject

```
@NetworkScope
abstract class NetworkComponent {
  @NetworkScope
  @Provides
  abstract fun api(): Api
}
```

В этом случае вопрос управления жизненным циклом той или иной ссылки ложится на разработчика. Вы можете задать обращение к типу как к синглтону, через аннотацию `@Singleton`; также поддерживается и lazy-инстанс (листинг 7.40).

Листинг 7.40. Lazy-обращение в Kotlin Inject

```
@Inject
 class NewsService

 @Inject
```

```
    class NewsUseCase(lazyNews: Lazy<NewsService>) {
        val newsService by lazyNews
    }
```

В целом это довольно легковесное и простое решение. Для установки достаточно объявить зависимость от основной библиотеки и `ksp` (листинги 7.41 и 7.42).

Листинг 7.41. Подключение Kotlin Inject

```
dependencies {
    ksp("me.tatarka.inject:kotlin-inject-compiler-ksp:*")
    implementation("me.tatarka.inject:kotlin-inject-runtime:*")
}
```

Листинг 7.42. Расширенное подключение KSP для Kotlin Inject

```
dependencies {
  // Генерация KSP MetaData для CommonMain
  kspCommonMainMetadata(libs.kotlinInject)

  // Генерация для всех таргетов
  kspAndroid("me.tatarka.inject:kotlin-inject-compiler-ksp:*")
  kspIosX64("me.tatarka.inject:kotlin-inject-compiler-ksp:*")
  kspIosArm64("me.tatarka.inject:kotlin-inject-compiler-ksp:*")
  kspIosSimulatorArm64("me.tatarka.inject:kotlin-inject-compiler-ksp:*")
}
```

Если вы хотите сохранить в проекте подключение в стиле Dagger, например, при миграции с Hilt/Dagger, то `Kotlin Inject` вам подойдет. Однако учтите, что это решение больше предназначено для работы с DI сугубо в commonMain, то есть когда вы реализуете общую архитектуру экранов в общем модуле. Объясняется это просто: аннотации Kotlin поддерживаются только на стороне Kotlin. Для нативного iOS-приложения этот функционал бесполезен.

7.6. Dagger Multiplatform

Kotlin Multiplatform по своей структуре и решениям больше близок миру Android, чем iOS. Поэтому библиотеки и инструментарий проектов Android постепенно включают поддержку Kotlin Multiplatform. Одним из ожидаемых решений является Dagger под KMP. Для этого нужна миграция с KAPT на KSP для кросс-платформенной обработки аннотаций и поддержка Kotlin Native. Несмотря на то что легковесный Koin уже давно предоставляет такую возможность, Dagger пока находится в процессе перехода.

KSP включен в версию Dagger 2.48 (август 2023), а в версии 2.51 процессинг аннотаций стал стабильным: https://github.com/google/dagger/releases.

> Complete Ksp support for Dagger Android: Added a Ksp Processor for Dagger Android ProguardProcessor that was previously missed. The ProguardProcessor is a Dagger Android implementation detail that makes sure the AndroidInjector

works correctly when shrinking tools obfuscate @ContributesAndroidInjector annotated injector class names.

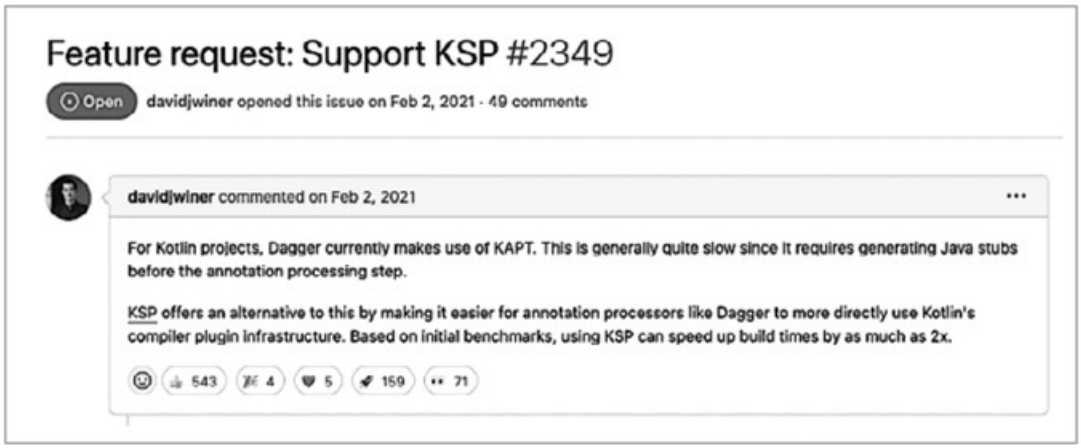


Рис. 7.6. Состояние поддержки KSP в Dagger

Правда, работает это только под Android, потому что Dagger зависит от JVM.

ПОДВЕДЕМ ИТОГИ

- ✓ Существует немало различных готовых решений для настройки DI в KMP-приложении.
- ✓ Настройка таких решений занимает мало времени и позволяет построить структуру довольно гибко.
- ✓ Решения постоянно дорабатываются и оптимизируются.

7.7. Доработка приложения

Вернемся к нашему приложению и внесем некоторые изменения.

1. Перенесем подключение зависимости в инициализаторы наших классов: сервиса, `UseCase`, `ViewModel` (листинги 7.43 и 7.44).

Листинг 7.43. Перенос зависимости из сетевого клиента в инициализатор

```
class NewsService(private val httpClient: NetworkClient) {

  suspend fun loadNews(): Result<NewsList> {
    return httpClient.request(URL)
  }

  companion object {
    val URL = "https://newsapi.org/v2/everything?q=science"
  }
}
```

Листинг 7.44. Перенос зависимости из сервиса в инициализатор UseCase

```
class NewsUseCase (private val newsService: NewsService): BaseUseCase<Unit,
NewsList?>() {
  override suspend fun execute(param: Unit): NewsList? {
    return newsService.loadNews()
  }
}
```

2. Подключим Koin актуальной версии и добавим `KoinComponent` (листинг 7.45).

Листинг 7.45. Настройка DI

```
class KoinDI : KoinComponent {

  //Модуль для сервисов. Синглтоны
  val serviceModule = module {
    single { NetworkClient() }
    single { NewsService(get()) }
  }

  // Модуль для UseCase. Синглтоны
  val usecaseModule = module {
    factory {
      NewsUseCase(get())
    }
  }

  //Модуль для ViewModel. Фабрика
  val vmModule = module {
    factory<NewsViewModel> { NewsViewModel(get()) }
  }

  //Создаем KoinApplication
  fun start() = startKoin {
    modules(listOf(serviceModule, usecaseModule, vmModule))
  }
}
```

Сервис и сетевой клиент у нас синглтоны, а `ViewModel` и `UseCase` будут создаваться с помощью фабрик при каждом обращении. Обратите внимание, что мы разбиваем подключение на модули.

3. Для удобства обращения добавляем фабрику с глобальной точкой входа и метод для получения инстанса по типу (листинг 7.46).

Листинг 7.46. Фабрика для работы с DI

```
object KoinDIFactory {
  val di by lazy {
    KoinDI().apply {
      start()
    }
  }
```

```
}

// Расширение для получения по типу
fun<T:Any> KoinDIFactory.resolve(clazz: KClass<*>):T? {
  return di.getKoin().get(clazz)
}
```

4. В iOSMain добавим служебные расширения для `ObjCClass` (см. в этой главе выше листинги 7.26, 7.30, 7.32).
5. На стороне iOS-приложения в NewsListVm заменим код для инициализации общей `ViewModel` (листинг 7.47).

Листинг 7.47. Подключение DI в ViewModel на стороне iOS

```
private lazy var vm: NewsViewModel? = {
        let vm = KoinDIFactory.shared.resolve(
                clazz_:  NewsViewModel.self) as? NewsViewModel
        vm?.newsFlow.collect(
                collector: itemsCollector,
                completionHandler: {_ in})
        return vm
}()
```

Если такой код кажется вам довольно громоздким, не расстраивайтесь. С переходом на общий UI на Compose мы от него избавимся и будем использовать только реализацию в общей `ViewModel`.

Аналогично подключаем общий DI для Android (листинг 7.48).

Листинг 7.48. Вызов ViewModel через DI в Android-приложении

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  val vm: NewsViewModel = KoinDIFactory.resolve(NewsViewModel::class)!!
  setContent {
    NewsListScreen(vm) { }
  }
}
```

Настало время расширять наше приложение. В следующей главе мы добавим поддержку еще одного общего решения Jetpack локального хранилища.

ГЛАВА 8

Храним данные в приложениях Kotlin Multiplatform

В этой главе рассмотрим:

- нативные хранилища iOS, Android;
- настройку работы с KMP DataStore;
- Room для Kotlin Multiplatform.

Хранение данных — один из важных аспектов реализации современного мобильного приложения. Он связан с обеспечением безопасности и затрагивает вопросы работы с чувствительными данными (например, информацией для авторизации), выполнения защищенных запросов и кэширования полученной ранее информации для экономии ресурсов.

Основные способы хранения данных в мобильном приложении — это:

- нативные хранилища Key-Value;
- хранилища с поддержкой баз данных (SQLite);
- хранение данных в виде файлов.

Мы уже написали небольшое кросс-платформенное приложение с полноценным бизнес-слоем и архитектурой, а также с выводом на нативный UI. Хотя в нем нет механизма авторизации и не нужно хранить чувствительные данные, нам может потребоваться хранить предыдущие результаты работы сетевых запросов а также данные, которые нужны только локально. Это будет небольшое хранилище для избранных новостей.

8.1. Нативные хранилища iOS и Android

Локальное хранение данных на устройстве использует нативные механизмы платформенных SDK. На Android это SharedPreferences, на iOS — UserDefaults. Эти хранилища используют механизм «ключ — значение» для сохранения и получения данных.

Попробуем создать на их основе общее решение для нашего общего модуля Kotlin Multiplatform.

Для начала посмотрим на код самих хранилищ. Для iOS на стороне нативного приложения мы можем использовать LocalStorage, как в листинге 8.1.

Листинг 8.1. Локальное хранилище на стороне iOS

```
class LocalStorage {

  // Singleton для удобства доступа к хранилищу
  static let shared = LocalStorage()

  private let userDefaults = UserDefaults.standard

  // Универсальный метод для сохранения данных
  func save<T>(_ value: T, forKey key: String) {
    userDefaults.set(value, forKey: key)
  }

  // Универсальный метод для получения данных
  func get<T>(forKey key: String, defaultValue: T) -> T {
    return userDefaults.object(forKey: key) as? T ?? defaultValue
  }

  // Удаление значения по ключу
  func removeValue(forKey key: String) {
    userDefaults.removeObject(forKey: key)
  }

  // Очистка всех данных
  func clearAllData() {
    if let bundleID = Bundle.main.bundleIdentifier {
      userDefaults.removePersistentDomain(forName: bundleID)
    }
  }
}
```

`UserDefaults` доступен в базовом iOS SDK как часть библиотеки `Foundation`. Значит, он будет доступен нам и в iOSMain в реализации на Kotlin Native. Перепишем наш `LocalStorage` на Kotlin (листинг 8.2).

Листинг 8.2. Локальное хранилище в iOSMain

```
class LocalStorage {

  private val userDefaults = NSUserDefaults.standardUserDefaults

  // Универсальный метод для сохранения данных
  fun <T : Any> save(value: T, key: String) {
    when (value) {
      is String -> userDefaults.setObject(value, forKey = key)
      is Int -> userDefaults.setInteger(value.toLong(), forKey = key)
```

```
      is Boolean -> userDefaults.setBool(value, forKey = key)
      is Double -> userDefaults.setDouble(value, forKey = key)
      is Float -> userDefaults.setFloat(value, forKey = key)
is ByteArray -> userDefaults.setObject(value, forKey = key)
      else -> throw IllegalArgumentException("Unsupported type")
    }
  }

  // Универсальный метод для получения данных
  fun <T> get(key: String, defaultValue: T): T? {
    return when (defaultValue) {
      is String -> userDefaults.stringForKey(key) as? T
      is Int -> userDefaults.integerForKey(key).toInt() as? T
      is Boolean -> userDefaults.boolForKey(key) as? T
      is Double -> userDefaults.doubleForKey(key) as? T
      is Float -> userDefaults.floatForKey(key) as? T
is ByteArray -> userDefaults.dataForKey(key) as? T
      else -> throw IllegalArgumentException("Unsupported type")
    }
  }

  // Удаление значения по ключу
  fun removeValue(key: String) {
    userDefaults.removeObjectForKey(key)
  }

  // Очистка всех данных
  fun clearAllData() {
    val bundleID = NSBundle.mainBundle.bundleIdentifier
    if (bundleID != null) {
      userDefaults.removePersistentDomainForName(bundleID)
    }
  }
}
```

`UserDefaults` на Swift позволяет использовать универсальный set, чтобы задать значение параметру любого типа. На Kotlin пришлось добавить обработку для всех простых типов. Для сложных объектов добавим поддержку сохранения в виде сериализованной строки (листинг 8.3).

Листинг 8.3. Доработка хранилища для iOS

```
private val json = Json { prettyPrint = true }


//Доработка метода сохранения
fun <T : Any> save(
    value: T, key: String,
    serializer: KSerializer<T>? = null
) {
    when (value) {
        //…
        (serializer != null) -> {
```

```
            val jsonString = json.encodeString(serializer!!, value)
            userDefaults.setObject(jsonString, forKey = key)
        }
    }
}

//Доработка получения данных
fun <T : Any> get(
    key: String,
    defaultValue: T,
    deserializer: KSerializer<T>? = null
): T? {
    return when (defaultValue) {
        //…
        else -> {
            // Десериализация объектов из JSON
            val jsonString = userDefaults.stringForKey(key)
            jsonString?.let {
                json.decodeFromString(deserializer!!, jsonString)
            } ?: defaultValue
        }
    }
}
```

Для кодирования и декодирования может использоваться JsonDecoder из главы 5 о сетевом взаимодействии.

Напишем теперь простое хранилище для Android на Kotlin (листинг 8.4).

Листинг 8.4. Хранилище для Android

```
class LocalStorage() {

  // Хранилище на SharedPreferences
  private var sharedPreferences: SharedPreferences? = null
  private var instance: LocalStorage? = null

// Метод для настройки хранилища
  fun getInstance(context: Context): LocalStorage {

    return instance ?: synchronized(this) {
      instance ?: LocalStorage().also {
        instance = it
        sharedPreferences = context.getSharedPreferences("MyPreferences",
                            Context.MODE_PRIVATE)
      }
    }
  }

  private val json = Json { prettyPrint = true } // Конфигурация JSON

  // Универсальный метод для сохранения данных
  fun <T : Any> save(value: T, key: String, serializer: KSerializer<T>) {
```

```
    sharedPreferences?.edit()?.apply {
      when (value) {
        is String -> putString(key, value).apply()
        is Int -> putInt(key, value).apply()
        is Boolean -> putBoolean(key, value).apply()
        is Float -> putFloat(key, value).apply()
        is Long -> putLong(key, value).apply()
        else -> {
          // Сериализация объектов в JSON и сохранение как строки
          val jsonString = json.encodeToString(serializer, value)
          putString(key, jsonString).apply()
        }
      }
    }
  }

  // Универсальный метод для получения данных
  fun <T : Any> get(key: String, defaultValue: T, serializer:
                          KSerializer<T>): T? {
    val sharedPreferences = sharedPreferences ?: return defaultValue
    return when (defaultValue) {
      is String -> sharedPreferences.getString(key, defaultValue) as T
      is Int -> sharedPreferences.getInt(key, defaultValue) as T
      is Boolean -> sharedPreferences.getBoolean(key, defaultValue) as T
      is Float -> sharedPreferences.getFloat(key, defaultValue) as T
      is Long -> sharedPreferences.getLong(key, defaultValue) as T
      else -> {
        // Десериализация объектов из JSON
        val jsonString = sharedPreferences.getString(key, null)
        jsonString?.let {
          json.decodeFromString(serializer, it)
        } ?: defaultValue
      }
    }
  }

  // Удаление значения по ключу
  fun removeValue(key: String) {
    sharedPreferences?.edit()?.remove(key)?.apply()
  }

  // Очистка всех данных
  fun clearAllData() {
    sharedPreferences?.edit()?.clear()?.apply()
  }
}
```

Наши хранилища отличаются внутренним механизмом, который используется для сохранения данных, и дополнительными методами настройки. В версии для Android (листинг 8.5) нам нужно передавать контекст приложения для инициализации `SharedPreferences`.

Листинг 8.5. Особенности подключения SharedPreferences

```
  // Хранилище на SharedPreferences
  private var sharedPreferences: SharedPreferences? = null
  private var instance: LocalStorage? = null

// Метод для настройки хранилища
  fun getInstance(context: Context): LocalStorage {

    return instance ?: synchronized(this) {
      instance ?: LocalStorage().also {
        instance = it
        sharedPreferences = context.getSharedPreferences("MyPreferences",
Context.MODE_PRIVATE)
      }
    }
  }
```

Теперь соединим наши практически нативные реализации для вызова из commonMain. Основные методы этих классов имеют идентичные сигнатуры, поэтому мы можем вынести их в декларацию `expect`-класса (листинг 8.6).

Листинг 8.6. Expect-класс хранилища

```
expect class LocalStorage() {
    fun <T : Any> save(value: T, key: String, serializer: KSerializer<T>? =
null)
    fun <T : Any> get(key: String, defaultValue: T, serializer: KSerializer<T>?
= null): T?
    fun removeValue(key: String)
    fun clearAllData()
}
```

Мы реализуем весь код в shared-модуле, поэтому интерфейс нам не нужен. Также у нас есть методы для Android с неподдерживаемой в общем коде сигнатурой. Поэтому оптимальным будет создать `actual`-реализации хранилищ для Android и iOS (листинг 8.7).

Листинг 8.7. Фрагмент actual-реализации для Android

```
actual class LocalStorage actual constructor(){

  // Настройки
  private var sharedPreferences: SharedPreferences? = null
  private var instance: LocalStorage? = null

  fun getInstance(context: Context): LocalStorage {

    return instance ?: synchronized(this) {
      instance ?: LocalStorage().also {
        instance = it
        sharedPreferences = context.getSharedPreferences("MyPreferences",
                            Context.MODE_PRIVATE)
```

```
      }
    }
  }

  private val json = Json { prettyPrint = true } // Конфигурация JSON

  // Универсальный метод для сохранения данных
  actual fun <T : Any> save(value: T, key: String, serializer:
                 KSerializer<T>?) {
    // Сохраняем
  }

  // Универсальный метод для получения данных
  actual fun <T : Any> get(key: String, defaultValue: T, serializer:
                 KSerializer<T>?): T? {
    // возвращаем значение
  }

  // Удаление значения по ключу
  actual fun removeValue(key: String) {
    sharedPreferences?.edit()?.remove(key)?.apply()
  }

  // Очистка всех данных
  actual fun clearAllData() {
    sharedPreferences?.edit()?.clear()?.apply()
  }
}
```

Еще один вариант объединения кода — это создание общего типа `Preferences` с `expect`-декларациями методов (листинг 8.8).

Листинг 8.8. Общий тип Preferences

```
expect class Preferences {

  fun saveString(value: String, key: String)
  fun getString(key: String): String
  fun saveBoolean(value: Boolean, key: String)
  fun getBoolean(key: String): Boolean
  fun saveInt(value: Int, key: String)
  fun getInt(key: String): Int
  fun saveFloat(value: Float, key: String)
  fun getFloat(key: String): Float
  fun saveObject(value: String, key: String)
  fun getObject(key: String): String?
  fun removeValue(key: String)
  fun clearAll()
}

// Создание экземпляра Preferences
expect fun createPreferences(): Preferences
```

В actual-реализациях методов для Android вызовем SharedPreferences (листинг 8.9).

Листинг 8.9. Preferences для Android, фрагмент реализации

```
actual class Preferences {

    var sharedPreferences: SharedPreferences? = null

    private var editor = sharedPreferences?.edit()

    actual fun saveString(value: String, key: String) {
        editor?.putString(key, value)?.apply()
    }

    actual fun getString(key: String): String {
        return sharedPreferences?.getString(key, "") ?: ""
    }
//...
}
```

В реализации для iOS используем UserDefaults (листинг 8.10).

Листинг 8.10. Фрагмент реализации Preferences для iOS

```
actual class Preferences {
    private val userDefaults = NSUserDefaults.standardUserDefaults

    actual fun saveString(value: String, key: String) {
        userDefaults.setObject(value, forKey = key)
    }

    actual fun getString(key: String): String {
        return userDefaults.stringForKey(key) ?: ""
    }
//...
}
```

Для androidMain потребуется также создать метод расширения Preferences, чтобы реализовать SharedPreferences с использованием контекста (листинг 8.11).

Листинг 8.11. Создание SharedPreferences с использованием контекста

```
fun Preferences.setup(context: Context) {
  sharedPreferences = context.getSharedPreferences("Settings",
                      Context.MODE_PRIVATE)
}
```

Модифицируем наше хранилище, чтобы вызывать методы Preferences (листинг 8.12).

Листинг 8.12. Общее хранилище

```
class CommonLocalStorage() : ILocalStorage {
```

```
  private val json = Json { prettyPrint = true }

  val preferences: Preferences = createPreferences()

  override fun <T : Any> save(value: T, key: String, serializer:
                    KSerializer<T>?) {
    when (value) {
      is String -> preferences.saveString(value, key = key)
      is Int -> preferences.saveInt(value, key = key)
      is Boolean -> preferences.saveBoolean(value, key = key)
      is Float -> preferences.saveFloat(value, key = key)
      (serializer != null) -> {
        val jsonString = json.encodeToString(serializer!!, value)
        preferences.saveObject(jsonString, key = key)
      }
    }
  }

  override fun <T : Any> get(key: String, defaultValue: T, serializer:
                                KSerializer<T>?): T? {
    return when (defaultValue) {
      is String -> preferences.getString(key) as? T
      is Int -> preferences.getInt(key).toInt() as? T
      is Boolean -> preferences.getBoolean(key) as? T
      is Float -> preferences.getFloat(key) as? T
      else -> {
        // Десериализация объектов из JSON
        val jsonString = preferences.getObject(key)
        jsonString?.let {
          json.decodeFromString(serializer!!, jsonString)
        } ?: defaultValue
      }
    }
  }

  override fun removeValue(key: String) {
    preferences.removeValue(key = key)
  }

  override fun clearAllData() {
    preferences.clearAll()
  }
}
```

Мы сохранили тот же код и ту же логику, что и в нативных хранилищах, но использовали `expect/actual`-реализацию `Preferences` для поддержки платформенного функционала.

Осталось добавить метод расширения для `CommonLocalStorage` в androidMain, чтобы передавать контекст (листинг 8.13).

Листинг 8.13. Метод расширения для настройки хранилища

```
fun CommonLocalStorage.setup(context: Context) {
    preferences.setup(context)
}
```

Потребуется продумать цепочку передачи контекста со стороны Android-приложения, чтобы поддержать интеграцию хранилища в общий модуль и общую логику.

8.2. DataStore в Kotlin Multiplatform

Мы можем упростить задачу и использовать готовое решение для работы с нативными хранилищами в Kotlin Multiplatform. Рассмотрим защищенное хранилище DataStore из Jetpack. Как и все библиотеки Jetpack, изначально данное решение было рассчитано только на Android; позже появилась поддержка кросс-платформы.

Решение на DataStore призвано заменить устаревший `SharedPreferences`, оно предоставляет более гибкий, безопасный и производительный способ работы с данными в формате или типизированными объектами. Хранилище поддерживает асинхронный и потокобезопасный API, позволяет избежать блокировки основного потока и улучшить производительность приложения.

Рассмотрим использование DataStore на специальных `Preferences`. Хранилище работает с данными в формате «ключ — значение» (аналогично `SharedPreferences`), не требует определения схемы данных и подходит для простых случаев, когда нужно хранить примитивные типы данных (строки, числа, булевы значения).

Для корректной работы DataStore нужно соблюдать следующие правила:

1. Никогда не создавайте более одного экземпляра DataStore для одного файла в одном процессе. Это может привести к поломке всех функций DataStore. Если к одному и тому же файлу хранилища обращаются несколько инстансов, DataStore выдаст исключение `IllegalStateException` при чтении или обновлении данных.
2. Общий тип DataStore должен быть неизменным.
3. Никогда не используйте одновременно `SingleProcessDataStore` и `MultiProcessDataStore` для одного и того же файла хранилища. Если вам необходимо поддержать доступ к DataStore из более чем одного процесса, используйте `MultiProcessDataStore`.

DataStore поддерживает KMP в версиях 1.1.0 и выше.

Чтобы настроить DataStore в вашем проекте KMP, добавьте зависимости для артефактов в файл `build.gradle.kts` для общего модуля:

- `androidx.datastore:datastore` — библиотека хранилища данных;

- `androidx.datastore:datastore-preferences` — библиотека хранилища данных настроек.

Листинг 8.14. Зависимости для DataStore

```
commonMain.dependencies {
     implementation("androidx.datastore:datastore-preferences-core:1.1.3")
     implementation("androidx.datastore:datastore-preferences:1.1.3")
}
```

Для создания хранилища нам потребуется описать создание DataStore на каждой платформе.

Для commonMain создадим такую функцию (листинг 8.15).

Листинг 8.15. Функция для создания DataStore в commonMain

```
fun createDataStore(producePath: () -> String): DataStore<Preferences> =
        PreferenceDataStoreFactory.createWithPath(
                produceFile = { producePath().toPath() }
        )

internal const val dataStoreFileName = "mydatastore.preferences_pb"
```

Для Android можно создать экземпляр DataStore с передачей ему контекста (листинг 8.16).

Листинг 8.16. Создание DataStore в androidMain

```
fun createDataStore(context: Context): DataStore<Preferences> = createDataStore(
    producePath = { context.filesDir.resolve(dataStoreFileName).absolutePath }
)
```

Вызовем общую функцию из commonMain и передадим в нее полученный путь к хранилищу.

На iOS потребуется использовать `NSFileManager`, чтобы задать путь для сохранения данных (листинг 8.17).

Листинг 8.17. Создание DataStore в iOSMain

```
fun createDataStore(): DataStore<Preferences> = createDataStore(
        producePath = {
            val documentDirectory: NSURL? = NSFileManager.defaultManager.
                                    URLForDirectory(
                    directory = NSDocumentDirectory,
                    inDomain = NSUserDomainMask,
                    appropriateForURL = null,
                    create = false,
                    error = null,
            )
            requireNotNull(documentDirectory).path + "/$dataStoreFileName"
        }
)
```

Используем функционал из commonMain и передадим в качестве параметра блок с вызовом получения пути к хранилищу через нативный код.

Вызывать создание хранилища полностью с нативной стороны неудобно, поэтому создадим общую обертку над DataStore в commonMain и добавим туда код для сохранения и чтения данных (листинг 8.18).

Листинг 8.18. Примерная реализация менеджера для DataStore

```
class CommonDataStore(
        private val store: DataStore<Preferences>
) {

    // Запись данных
    suspend fun saveData(key: String, value: String) {
        store.edit { preferences ->
            preferences[stringPreferencesKey(key)] = value
        }
    }

    // Чтение данных
    val data: Flow<String?> = store.data
            .map { preferences ->
                preferences[stringPreferencesKey("key")]
            }
}
```

DataStore для iOSMain/androidMain можно будет передавать со стороны нативной части.

Такие хранилища подойдут для хранения простых примитивных типов. Для хранения более сложных объектов нужны продвинутые механизмы, например локальные базы данных.

8.3. Room

Если нам нужно сохранять локально большие объемы данных и/или сложные и структурированные объекты, то вместо хранилища на SharedPrerefences стоит использовать локальную базу данных.

Основным рекомендуемым решением для создания хранилища является библиотека для хранения данных Room из набора Jetpack. Первоначально библиотека поддерживала только Android-решения. В мае 2024 года Google представил версию Room для кросс-платформенных приложений Kotlin Multiplatform.

Стоит упомянуть, что Google рекомендует использовать Room вместо прямого обращения к API SQLite.

Рассмотрим работу с Room на примере небольшого приложения для сохранения списка дел Todo.

8.3.1. Установка и подключение Room

Начнем с настроек проекта. Чтобы настроить Room в проекте KMP, потребуется установить библиотеку Room, SQLite и следующие зависимости:

- `androidx.room:room-gradle-plugin` — плагин Gradle для настройки схем комнат;
- `androidx.room:room-compiler` — процессор KSP, генерирующий код;
- `androidx.room:room-runtime` — часть библиотеки времени выполнения;
- `androidx.sqlite:sqlite-bundled` — встроенная библиотека SQLite (не обязательно).

SQLite — это хранилище по умолчанию, которое мы используем под капотом Room.

> **ВНИМАНИЕ!** Не все версии Room поддерживают Kotlin Multiplatform. Текущая версия Room с поддержкой кросс-платформенной работы — 2.7.0-alpha01 или выше.

Пропишем зависимость в каталог lib.versions (листинг 8.19).

Листинг 8.19. Зависимости для Room в lib.versions

```
/*lib.versions*/
[versions]
androidxRoom = "2.7.0-alpha01"
sqlite = "2.5.0-alpha01"

[libraries]
androidx-room-compiler = {
  group = "androidx.room",
  name = "room-compiler",
  version.ref = "androidxRoom"
}
androidx-room-runtime = {
  group = "androidx.room",
  name = "room-runtime",
  version.ref = "androidxRoom"
}
sqlite-bundled = {
  module = "androidx.sqlite:sqlite-bundled",
  version.ref = "sqlite"
}
sqlite = {
  module = "androidx.sqlite:sqlite",
  version.ref = "sqlite"
}
```

Теперь нам нужно подключить плагин для Room в `build.gradle.kts` (листинг 8.20).

Листинг 8.20. Подключение Room как плагина

```
/*lib.versions*/
[plugins]
room = { id = "androidx.room", version.ref = "androidxRoom" }

/*build.gradle.kts app*/
plugins {
  alias(libs.plugins.room).apply(false)
}

/*build.gradle.kts shared*/
plugins {
  alias(libs.plugins.room)
}
```

Подключим зависимости также в commonMain dependencies в `build.gradle.kts` нашего shared-модуля (листинг 8.21).

Листинг 8.21. Подключение зависимостей к build.gradle.kts

```
sourceSets {
    commonMain.dependencies {
        implementation(libs.androidx.room.runtime)
        implementation(libs.sqlite.bundled)
        implementation(libs.sqlite)
    }
}
```

Для генерации данных предыдущие версии Room использовали механизм KAPT (Kotlin Annotation Processing Tool). Одним из основных этапов миграции Room для настройки поддержки мультиплатформы был переход с KAPT на технологию KSP (Kotlin Symbol Processing). Поэтому нам надо добавить KSP как зависимость и подключить ее в качестве плагина (листинг 8.22).

Листинг 8.22. Подключение KSP

```
/*lib.versions*/
[versions]
  ksp = "1.9.23"
  kotlin = " 1.9.23-1.0.19"

[plugins]
  ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

ВАЖНО! Учтите, что версия Kotlin должна совпадать с мажорной версией KSP.

В текущем примере используется версия 1.9.23 для поддержки совместимости с другими технологиями, использованными в проекте.

Подключим KSP как плагин в `build.gradle.kts` приложения и блок плагинов shared-модуля (листинг 8.23).

Листинг 8.23. Подключение KSP

```
/*build.gradle.kts app*/
plugins {
  //...
  alias(libs.plugins.ksp) apply false
}

/*build.gradle.kts shared*/
plugins {
  //...
  id("com.google.devtools.ksp")
}
```

Также добавим блок процессинга компилятора Room через KSP для всех поддерживаемых таргетов в самый низ `build.gradle.kts` (shared) (листинг 8.24).

Листинг 8.24. Настройка таргетов KSP

```
dependencies {
    add("kspAndroid", libs.androidx.room.compiler)
    add("kspIosSimulatorArm64", libs.androidx.room.compiler)
    add("kspIosX64", libs.androidx.room.compiler)
    add("kspIosArm64", libs.androidx.room.compiler)
}
```

Подробнее о KSP и работе с ним для упрощения и оптимизации кода приложения мы поговорим в главе 11 об инструментах оптимизации.

> **ПРЕДУПРЕЖДЕНИЕ** При использовании Kotlin версии 1.9.x необходимо добавить свойство `kotlin.native.disableCompilerDaemon = true` в файл конфигурации `gradle.properties`, чтобы процессор KSP Room работал правильно.

Это свойство не требуется при использовании Kotlin версии 2.0 или выше. Для получения дополнительной информации см. https://youtrack.jetbrains.com/issue/KT-6576 (листинг 8.25).

Листинг 8.25. Настройка для Kotlin 1.9

```
kotlin.native.disableCompilerDaemon = true.

# Disabled due to https://youtrack.jetbrains.com/issue/KT-65761
kotlin.native.disableCompilerDaemon = true
```

Для корректной работы необходимо указать Room путь к схемам базы данных:

```
room {
    schemaDirectory("$projectDir/schemas")
}
```

Схема Room определяет структуру самой базы данных, используемых таблиц и моделей данных, связи между объектами и т. п. Вручную ее прописывать не надо, она сгенерируется при компиляции нашего приложения.

Синхронизируем Gradle. Готово, мы установили Room.

8.3.2. Настройка хранилища Room

Теперь давайте настроим само хранилище. Так же как и в Android-приложении, нам потребуется выполнить следующие шаги, хотя и с некоторыми нюансами:

1. Создать модель данных Entity для таблицы базы данных. Таблица будет сгенерирована во время компиляции.
2. Создать Dao для запросов из этой таблицы. На его основе будут сгенерированы методы-запросы ко всем нашим таблицам.
3. Создать и настроить хранилище как класс-наследник RoomDatabase.
4. Создать репозиторий для запросов — шаг опциональный, необходимый больше для соблюдения архитектурного порядка.

Для модели данных используем обычный data class с нужными нам полями (листинг 8.26).

Листинг 8.26. Модель данных для контента

```
data class TodoItem(
        val id: Long = 0,
        val title: String,
        val content: String,
        val date: String
)
```

Для того чтобы показать Room, что из этого класса нужно сгенерировать таблицу базы данных, пометим его специальной аннотацией `@Entity` (листинг 8.27).

Листинг 8.27. Модель данных Entity

```
@Entity
data class TodoEntity(
        @PrimaryKey(autoGenerate = true) val id: Long = 0,
        val title: String,
        val content: String,
        val date: String
)
```

Для обозначения поля первичного ключа используется аннотация `@PrimaryKey`. Параметр `autogenerate` гарантирует, что ключ будет генерироваться автоматически.

Чтобы разделить слои представления и данных, мы будем использовать два класса с одинаковыми полями и добавим расширения для их преобразования друг в друга (листинг 8.28).

Листинг 8.28. Методы преобразования между сущностями

```
fun TodoEntity.toItem(): TodoItem = TodoItem(id, title, content, date)

fun TodoItem.toEntity(): TodoEntity = TodoEntity(title, content, date)
```

Такой подход годится, если data-класс используется в модулях, куда специализированные библиотеки (в нашем случае Room) подключать не нужно.

Теперь добавим интерфейс Dao и опишем методы для операций добавления элемента (Insert) и получения данных (Select), как в листинге 8.29.

Листинг 8.29. Dao для хранилища

```
@Dao
interface TodoDao {

    @Insert
    suspend fun insert(item: TodoEntity)

    @Query("SELECT count(*) FROM TodoEntity")
    suspend fun count(): Int

    @Query("SELECT * FROM TodoEntity")
    fun getAllAsFlow(): Flow<List<TodoEntity>>
}
```

Методы для добавления пометим аннотацией `@Insert`. Для генерации кода методов запросов используется аннотация `@Query`, в которую передается строка с SQL-запросом к нашим таблицам данных.

Версия Room для KMP поддерживает использование `expect/actual` для создания разных Dao под разные таргеты.

Перейдем к созданию и настройке самой базы данных. Так же как и для Android-версии, создадим абстрактный класс-наследник `RoomDatabase` (листинг 8.30).

Листинг 8.30. Конструктор базы данных

```
@Database(entities = [TodoEntity::class], version = 1)
@ConstructedBy(AppDatabaseConstructor::class)
abstract class AppDatabase : RoomDatabase() {
  abstract fun getDao(): TodoDao
}
```

Далее необходимо добавить `expect`-декларацию для конструктора базы данных (листинг 8.31).

Листинг 8.31. Конструктор базы данных

```
// The Room compiler generates the `actual` implementations.
@Suppress("NO_ACTUAL_FOR_EXPECT")
expect object AppDatabaseConstructor : RoomDatabaseConstructor<AppDatabase> {
    override fun initialize(): AppDatabase
}
```

Actual-реализации `RoomDatabaseConstructor` нам создавать не нужно, компилятор Room сгенерирует их сам. Чтобы игнорировать предупреждение `"Expected object 'AppDatabaseConstructor' has no actual declaration in module"`, пометим метод инициализации аннотацией `@Suppress("NO_ACTUAL_FOR_EXPECT")`.

Следующим шагом необходимо настроить драйвер SQLite для Room. Эти драйверы различаются в зависимости от целевой платформы: разный API файловых систем разных платформ определяет местоположение хранилища по-разному. Например, в Android местоположение базы данных обычно получается через API `Context.getDatabasePath()`, а в iOS местоположение базы данных — с помощью NSFileManager.

Для настройки драйвера создаем специальные билдеры-строители `RoomDataBase.Builder` через методы `expect/actual` (листинг 8.32).

Листинг 8.32. Строитель базы данных для Android

```
//Android
fun getDatabaseBuilder(ctx: Context): RoomDatabase.Builder<AppDatabase> {
    val appContext = ctx.applicationContext
    val dbFile = appContext.getDatabasePath("my_room.db")
    return Room.databaseBuilder<AppDatabase>(
        context = appContext,
        name = dbFile.absolutePath
    )
}
.setDriver(BundledSQLiteDriver())
.setQueryCoroutineContext(Dispatchers.IO)


fun getDatabase(ctx: Context): AppDatabase {
    return getDatabaseBuilder(ctx).build()
}
```

Для получения пути к хранилищу потребуется передать контекст приложения.

Для iOS, чтобы создать экземпляр базы данных, нужно указать путь к базе данных с помощью `NSFileManager`. Обычно такой контент располагают в `NSDocumentDirectory` (листинг 8.33).

Листинг 8.33. Строитель для базы данных iOS

```
//iOS
fun getDatabaseBuilder(): RoomDatabase.Builder<AppDatabase> {
    val dbFilePath = documentDirectory() + "/my_room.db"
```

```
    return Room.databaseBuilder<AppDatabase>(
        name = dbFilePath,
    )
}

private fun documentDirectory(): String {
  val documentDirectory = NSFileManager.defaultManager.URLForDirectory(
    directory = NSDocumentDirectory,
    inDomain = NSUserDomainMask,
    appropriateForURL = null,
    create = false,
    error = null,
  )
  return requireNotNull(documentDirectory?.path)
}

fun getDatabase(): AppDatabase {
    return getDatabaseBuilder().build()
}
```

В предыдущих фрагментах кода используется `BundledSQLiteDriver`. Это рекомендуемый драйвер, включающий SQLite, скомпилированный из исходного кода, который обеспечивает наиболее согласованную и актуальную версию SQLite на всех платформах. Если вы хотите попробовать SQLite, предоставляемый ОС, нужно использовать API setDriver в исходных наборах для конкретной платформы, в которых указан драйвер для выбранной платформы. Для Android вы можете использовать `AndroidSQLiteDriver`, а для iOS — `NativeSQLiteDriver`. Чтобы подключить NativeSQLiteDriver, потребуется настроить компоновщик в build.gradle.kts общего модуля и добавить в него опцию линковки — так iOS-приложение сможет динамически связываться с системным SQLite (листинг 8.34).

Листинг 8.34. Настройка компоновщика для iOS

```
// shared/build.gradle.kts

kotlin {
  listOf(
    iosX64(),
    iosArm64(),
    iosSimulatorArm64()
  ).forEach { iosTarget ->
    iosTarget.binaries.framework {
      baseName = "TodoApp"
      isStatic = true
      // Required when using NativeSQLiteDriver
      linkerOpts.add("-lsqlite3")
    }
  }
}
```

У наших билдеров разная сигнатура, поэтому пометить их `actual` и задать общую сигнатуру с `expect` мы не можем. Решим эту проблему следующим образом: используем Koin для инициализации хранилища и создадим `expect/actual` модуль (листинг 8.35).

Листинг 8.35. Expect/actual-реализация модуля для хранилищ в Koin

```
//commonMain
expect fun platformModule(): Module

//androidMain
actual fun platformModule() = module {
    single<AppDatabase> { getDatabase(get()) }
}

//iOSMain
actual fun platformModule() = module {
    single<AppDatabase> { getDatabase() }
}
```

Также мы можем модифицировать метод для создания нашей базы данных, чтобы через DI передавать инстанс нашего строителя хранилища (листинг 8.36).

Листинг 8.36. Общий метод создания базы данных

```
// shared/src/commonMain

fun getRoomDatabase(
    builder: RoomDatabase.Builder<AppDatabase>
): AppDatabase {
  return builder
      .addMigrations(MIGRATIONS)
      .fallbackToDestructiveMigrationOnDowngrade()
      .setDriver(BundledSQLiteDriver())
      .setQueryCoroutineContext(Dispatchers.IO)
      .build()
}
```

Регистрация DI тоже изменится (листинг 8.37).

Листинг 8.37. Expect/actual-реализация платформенного модуля

```
//commonMain
expect fun platformModule(): Module

//androidMain
actual fun platformModule() = module {
  single<AppDatabase> { getDatabase() }
  single<RoomDatabase.Builder<AppDatabase>> {
    getDatabaseBuilder(get())
  }
}
```

```
//iOSMain
actual fun platformModule() = module {
  single<AppDatabase> { getDatabase() }
  single<RoomDatabase.Builder<AppDatabase>> {
    getDatabaseBuilder()
  }
}
```

Теперь самое сложное: нам надо как-то передать контекст со стороны Android-приложения. Когда мы реализовывали хранилище «ключ — значение», у нас уже возникала проблема с финальным решением.

Воспользуемся возможностями DI Koin и напишем функцию в commonMain с параметром-блоком (листинг 8.38).

Листинг 8.38. Инициализация Koin с поддержкой KoinAppDeclaration

```
fun initKoin(appDeclaration: KoinAppDeclaration = {}) =
    startKoin {
        appDeclaration()
        modules(platformModule())
    }
```

`KoinAppDeclaration` позволит нам получить доступ к контексту приложения.

При работе с DI мы используем самописную фабрику для упрощения доступа и получения инстансов. Изменим метод настройки, добавив туда передачу блока для создания `KoinAppDeclaration` (листинг 8.39).

Листинг 8.39. Доработка фабрики DI для поддержки KoinAppDeclaration

```
object Koin {
   var di: KoinApplication? = null

   fun setupKoin(appDeclaration: KoinAppDeclaration = {}) {
       if (di == null) {
           di = initKoin(appDeclaration)
       }
   }
}
```

По умолчанию передается пустой блок, чтобы мы могли безопасно использовать решение на стороне iOS.

Для завершения настройки вызовем из нативных Android- и iOS-приложений setupKoin (листинг 8.40).

Листинг 8.40. Передача контекста приложения в Koin

```
//Android
Koin.setupKoin {
    androidContext(applicationContext)
}
```

```
//iOS
Koin.setupKoin()
```

8.3.3. Работа с хранилищем

Наконец мы закончили с инициализациями и настройками. Теперь подключим логику работы с хранилищем к экранам приложения.

Добавим репозиторий, где вызовем методы Dao (листинг 8.41).

Листинг 8.41. Репозиторий для работы с хранилищем

```
class TaskRepository(
        private val database: AppDatabase
) {
private val dao: TodoDao by lazy {
    database.getDao()
}

    suspend fun addTodo(todoEntity: TodoEntity) {
        dao.insert(todoEntity)
    }

    suspend fun loadTodos(): Flow<List<TodoEntity>> {
        return dao.getAllAsFlow()
    }
}
```

Добавим в наши `ViewModel` функции вызова репозитория и его методов. Представьте, что у нас есть простой экран с текстовым полем и мы хотим сохранять текст, который введет в него пользователь. Для связи между свойством `ViewModel` и UI мы будем использовать механизм `MutableStableFlow`. Метод сохранения значения строки в репозиторий будем вызывать кликом по кнопке (листинг 8.42).

Листинг 8.42. Вызов создания записи из ViewModel

```
class AddTodoViewModel(
        private val taskRepository: TaskRepository
) : ViewModel() {

    // Свойство для хранения текста из текстового поля
    val titleText: MutableStateFlow<String>
            = MutableStateFlow<String>("")

    fun onConfirm() {
        // Вызываем сохранение
        viewModelScope.launch {
            taskRepository.addTodo(TodoEntity(title = titleText.value))
        }
    }
}
```

Во `ViewModel` экрана, на котором будем выводить наши `Todo`, добавим метод для загрузки данных. Google рекомендует реализовывать получение данных из хранилища Room как `Flow<*>`. Подпишемся на результат через `collectLatest` (листинг 8.43).

Листинг 8.43. Подключение получения данных из репозитория

```
class TodoViewModel(
  private val repository: TaskRepository
) : ViewModel() {

  val tasks: MutableSharedFlow<List<TodoEntity>> =
    MutableSharedFlow(1, onBufferOverflow = BufferOverflow.DROP_OLDEST)

  fun loadData() {
    viewModelScope.launch {
      repository.loadTodos().collectLatest {
        tasks.tryEmit(it)
      }
    }
  }
}
```

Готово. Теперь можно подключить код к своему UI и проверить работу. Полный код примера вы найдете по ссылке в конце книги.

Если вы мигрируете с существующего Android-приложения, где у вас уже было хранилище на Room, то, кроме настройки второго таргета и подключения контекста, сложностей у вас никаких не будет. Но есть и ограничения для версии Kotlin Multiplatform, которые могут оказаться критическими.

8.3.4. Ограничения Room KMP

Использование в не-Android-таргетах методов, помеченных аннотацией `@RawQuery`, вызовет ошибку.

Функции, помеченные `@RawQuery` и скомпилированные для платформ, отличных от Android, должны будут использовать параметр типа `RoomRawQuery` вместо `SupportSQLiteQuery` (листинг 8.44).

Листинг 8.44. Пример RoomRawQuery

```
@Dao
interface TodoDao {
    @RawQuery
    suspend fun getTodos(query RoomRawQuery): List<TodoEntity>
}
```

Кроме этого, `RoomRawQuery` можно использовать для создания запроса во время выполнения (листинг 8.45).

Листинг 8.45. Пример RoomRawQuery

```
suspend fun getTodosWithLowercaseTitle(
  title: String
): List<TodoEntity> {

  val query = RoomRawQuery(
    sql = "SELECT * FROM TodoEntity WHERE title = ?"
    onBindStatement = {
      it.bindText(1, title.lowercase())
    }
  )
  return todosDao.getTodos(query)
}
```

Поддержка этой аннотации запланирована к добавлению в одной из следующих версий Room.

Также поддерживаются только в Android:

1. API обратного вызова коллбэка:
   - `RoomDatabase.Builder.setQueryCallback`;
   - `RoomDatabase.QueryCallback`.
2. Автоматическое закрытие базы данных по тайм-ауту:
   - `RoomDatabase.Builder.setAutoCloseTimeout`.
3. Множественные экземпляры хранилища:
   - `RoomDatabase.Builder.enableMultiInstanceInvalidation`.

   Обратите внимание, что у вас должен быть только один инстанс хранилища. Обязательно зарегистрируйте его как синглтон в вашем DI.
4. Создание базы данных из ассетов, файлов и т. п.:
   - `RoomDatabase.Builder.createFromAsset`;
   - `RoomDatabase.Builder.createFromFile`;
   - `RoomDatabase.Builder.createFromInputStream`;
   - `RoomDatabase.PrepackagedDatabaseCallback`.

ПОДВЕДЕМ ИТОГИ

- ✓ В качестве легковесного хранилища можно использовать собственную реализацию на базе нативных хранилищ.
- ✓ Jetpack предоставляет решение DataStore для простой работы с нативными механизмами хранения Android и iOS.
- ✓ Для хранения больших объемов и/или структурированных данных используйте локальную базу данных на Room. Прямое использование SQLite API не рекомендовано.

8.4. Применение на практике

У нас есть список новостей. Давайте дадим пользователям возможность добавлять некоторые новости в закладки. В качестве решения будем использовать хранилище на Room.

1. Нам потребуется добавить новые поля в модель данных (листинг 8.46): `isFavorite` — флаг избранности.

Листинг 8.46. Исходная модель

```
@kotlinx.serialization.Serializable
data class NewsItem(
        @SerialName("author") val author: String?,
        val title: String?,
        val description: String?,
        val url: String,
        val urlToImage: String?,
        val publishedAt: String?,
        val content: String?,
        var isFavorite: Boolean = false
)
```

Создадим Entity для хранения этой модели (листинг 8.47).

Листинг 8.47. Entity для новости

```
@Entity("news")
data class NewsItemEntity(
    val author: String?,
    val title: String?, val description: String?,
    @PrimaryKey(autoGenerate = false) val url: String, // ключ
    val urlToImage: String?,
    val publishedAt: String?,
    val content: String?,
    var isFavorite: Boolean = false
)
```

В качестве первичного ключа и идентификатора будем использовать url.

2. Подключим и настроим Room, как в листингах в разделе 8.3.
3. Добавим Dao для запросов (листинг 8.48).

Листинг 8.48. Dao

```
@Dao
interface NewsListDao {
    @Insert
    suspend fun insert(item: NewsItemEntity)

    @Query("SELECT count(*) FROM news")
    suspend fun count(): Int
```

```
    @Query("SELECT * FROM news")
    suspend fun getAll(): List<NewsItemEntity>

    @Update
    suspend fun updateItem(item: NewsItemEntity)
}
```

Также потребуются запросы на обновление, получение и сохранение элементов.

4. Создадим репозиторий для обращения к Dao через базу данных (листинг 8.49).

Листинг 8.49. Репозиторий для работы с новостями

```
class NewsRepository constructor(
    private val database: AppDatabase
) {

    private val dao: NewsListDao by lazy {
        database.getDao()
    }

    suspend fun getAll(): List<NewsItemEntity> {
        return dao.getAll()
    }

    suspend fun save(item: NewsItemEntity) {
        val all = getAll()
// Если нет объекта, добавляем
        if (all.find { it.url == item.url } == null) {
            dao.insert(item)
        } else {
// Обновляем существующий объект
            update(item)
        }
    }

    suspend fun saveAll(items: List<NewsItemEntity>) {
        items.forEach { save(it) }
    }

    suspend fun update(item: NewsItemEntity) {
        dao.updateItem(item)
    }
}
```

5. Для удобства работы можно дополнительно обернуть запросы к репозиторию в отдельные UseCase (листинг 8.50).

Листинг 8.50. Дополнительные UseCase (возможная реализация)

```
// Загружаем данные
class NewsCachedUseCase(private val repository: NewsRepository):
                                    BaseUseCase<Unit, List<NewsItem>>() {
```

```
    override suspend fun execute(param: Unit): List<NewsItem> {
        return repository.getAll().map { it.toItem() }
    }
}

// Сохраняем
class SaveNewsUseCase(private val repository: NewsRepository):
BaseUseCase<NewsItem?, Unit>() {

    override suspend fun execute(param: NewsItem?): Unit {
        if (param != null) {
            repository.save(param.toEntity())
        }
    }
}
```

6. Затем нужно подключить логику к `ViewModel`. Добавим `UseCase` в параметры `ViewModel` (листинг 8.51).

Листинг 8.51. Добавление UseCase в параметры

```
class NewsViewModel(
    private val useCase: NewsUseCase,
    private val loadCase: NewsCachUseCase,
    private val saveCase: SaveNewsUseCase
) : BaseViewModel() {/***/}
```

7. Нужно добавить сохранение получаемых элементов по сети, но с учетом того, что они могут повторять существующие в хранилище элементы. Потребуется сохранять значение isFavorite. Модифицируем код метода загрузки (листинг 8.52).

Листинг 8.52. Метод загрузки данных с сохранением в кэш

```
fun loadNews() {
    viewModelScope.launch {
        // Кэшированные данные
        val cachedResult = loadCase.invoke(Unit)
        cachedItems.addAll(cachedResult.getOrNull().orEmpty())

        val result = useCase.invoke(Unit)
        result.getOrNull()?.let { newsList ->
            // Временный список
            val temp = mutableListOf<NewsItem>()
            // Обрабатываем статьи в обратном порядке, чтобы последние
               статьи были первыми
            newsList.articles.orEmpty().reversed().forEach { article ->
                // Берем только те статьи, которых нет в кеше
                if (cachedItems.find { it.url == article.url } == null) {
                    temp.add(article)
                }
            }
            // Добавляем в кэш в нужном порядке
```

```
            cachedItems.addAll(temp.reversed())
            cachedItems.forEach {
                saveCase.invoke(it)
            }
            // Выводим
            withContext(Dispatchers.Main) {
                newsFlow.tryEmit(cachedItems)
            }
        }
    }
}
```

8. Теперь добавим сам метод добавления/удаления элемента в избранное (листинг 8.53).

Листинг 8.53. Добавление/удаление элемента в избранное

```
fun onFavoriteClick(item: NewsItem) {
// Создаем копию элемента, так как у нас data-класс
    val newsItem = item.copy(isFavorite = !item.isFavorite)
    val index = cachedItems.indexOf(item)
    // Обновляем элемент по индексу в списке кэша
    cachedItems[index] = newsItem
    viewModelScope.launch(Dispatchers.IO) {
        // Сохраняем в базе данных
        saveCase.invoke(newsItem)
        withContext(Dispatchers.Main) {
            newsFlow.tryEmit(cachedItems)
        }
    }
}
```

Код вызова события из View оставим до главы 10, где мы рассмотрим работу с общим UI в Kotlin Mulitplatform. Сейчас можно добавить какую-нибудь кнопку в элемент списка и пробросить клик по ней до `ViewModel`.

Получим примерно такой результат (рис. 8.1).

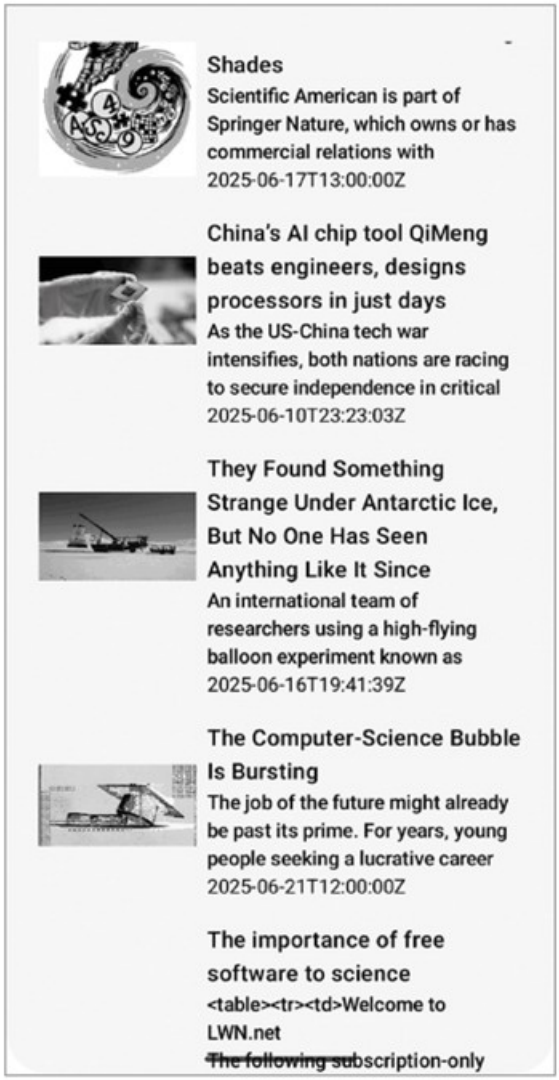


Рис. 8.1. Пример работы приложения с реализацией функции добавления в избранное

В следующей главе рассмотрим, как проверять код на работоспособность и надежность с помощью покрытия тестами.

ГЛАВА 9

Тестируем приложение Kotlin Multiplatform

В этой главе рассмотрим:

- настройку тестов в KMP-приложении;
- тестирование Kotlin Test и Kotest;
- общие и нативные тесты.

Тестирование в Kotlin Multiplatform — это важная часть разработки кросс-платформенных приложений, позволяющая убедиться, что код работает корректно на всех поддерживаемых платформах (например, Android, iOS, JVM, JS и т. д.). Правильное и полное покрытие приложения тестами помогает удостовериться не только в его работоспособности в целом, но и в том, что задачи выполняются правильно и безопасно. Также тесты — это своего рода способ документации функционала приложения и передачи базы знаний.

Несмотря на то что тесты очень важны, довольно часто задачу покрытия кода тестами выполняют по остаточному принципу, больше внимания уделяя реализации кода. Однако в настоящее время наметилась обратная тенденция.

Поскольку мы учимся делать правильно работающие приложения Kotlin Multiplatform со сложным функционалом, давайте рассмотрим, как писать тесты, и попробуем применить их к нашему коду.

Тесты для мультиплатформенных проектов можно разделить на две категории:

1. Тесты для общего кода. Их можно запускать на любой платформе с использованием любой поддерживаемой среды разработки.
2. Тесты для платформо-специфичного кода. Эти тесты необходимы для проверки логики, характерной для конкретной платформы. Для их реализации используется соответствующий платформенный фреймворк, включая его дополнительные функции, более богатый API и широкий набор правил. Сейчас обе категории поддерживаются в мультиплатформенных проектах.

В настоящее время для тестирования кода приложений используются следующие фреймворки:

- Kotlin Test (решение для commonMain с расширениями для androidMain и iOSMain, подробнее в разделе 9.1);
- Junit (только JVM и Android);
- Kotest Multiplatform (раздел 9.5);
- Junit/Kotest для Android, Xcode Unit для iOS.

9.1. Kotlin Test

Начнем с базового и основного решения для тестирования кода в Kotlin Multiplatform.

Нам потребуется создать специальный таргет, чтобы работать с тестами для каждого таргета, который мы хотим ими покрыть. Для этого выберем создание новой директории для shared-модуля (рис. 9.1).

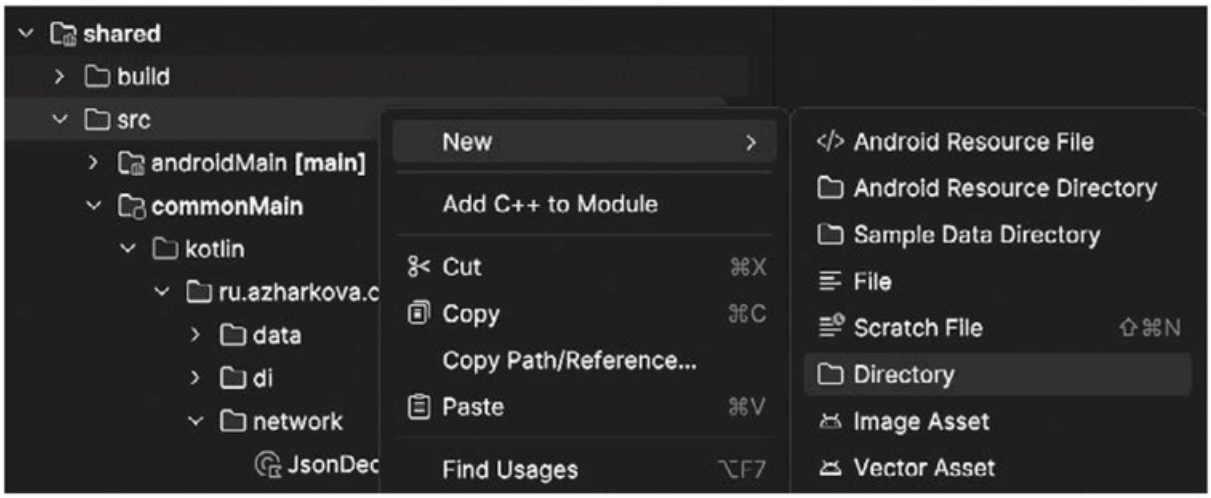


Рис. 9.1. Добавление новой директории

Начнем вводить в названии директории commonTest. Выберем из предложенных вариантов commonTest/kotlin (рис. 9.2).

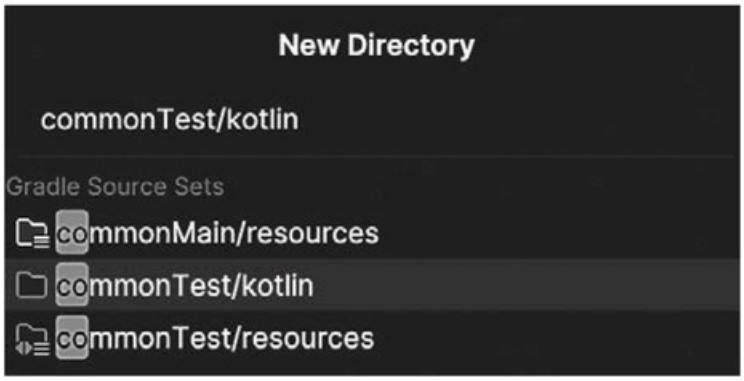


Рис. 9.2. Выбор директории commonTest/kotlin

Добавим новый таргет в описание `build.gradle.kts` shared-модуля и подключим библиотеку Kotlin Test (листинг 9.1).

Листинг 9.1. Подключение библиотеки Kotlin Test

```
// Новый таргет в build.gradle.kts

    commonTest.dependencies {
        implementation(libs.kotlin.test)
    }

// Библиотека из каталога gradle
[libraries]
kotlin-test = { module = "org.jetbrains.kotlin:kotlin-test",
version.ref = "kotlin" }
```

В последних версиях Kotlin Multiplatform эта библиотека внесена в каталог по умолчанию. Остается только подключить ее к нужному таргету.

Нажмем кнопку синхронизации и перейдем к следующему шагу. Добавим тест: создадим пустой файл Kotlin с простой функцией (листинг 9.2). Для того чтобы функция работала как тест, пометим ее аннотацией `@Test`.

Листинг 9.2. Пример теста

```
@Test
fun sampleTest() {

}
```

Для того чтобы запустить тестовую функцию, надо поместить ее внутрь класса (рис. 9.3).

```
1     import kotlin.test.Test
2
3     class SampleTest {
4
5         @Test
6         fun performTest() {}
7     }
```

Рис. 9.3. Правильная декларация теста

Если нажать на стрелку слева от функции, появится меню для запуска определенного теста (рис. 9.4).

Нажмем на стрелку возле класса, после этого появится меню для запуска тестов на весь класс (рис. 9.5).

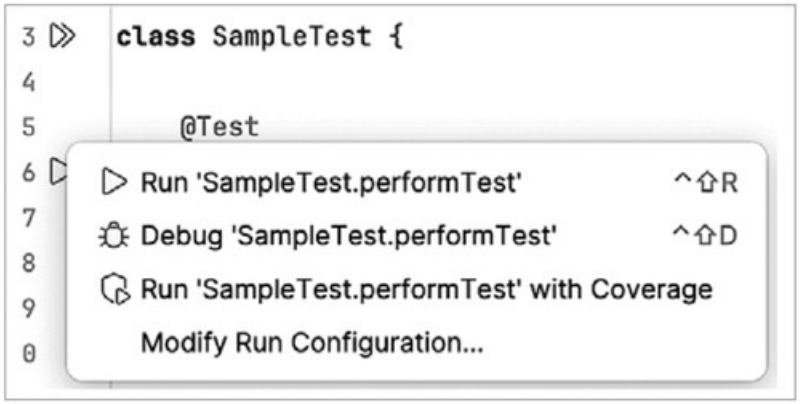


Рис. 9.4. Запуск тестовой функции

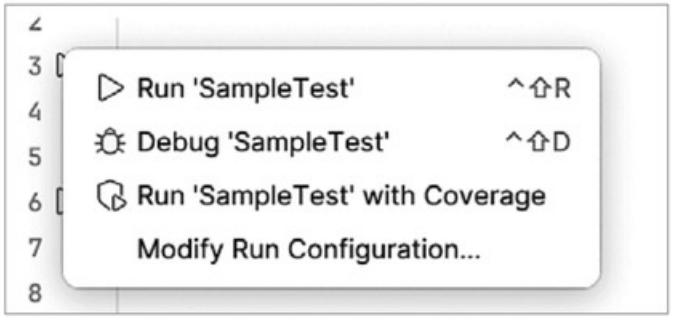


Рис. 9.5. Пример запуска теста

Выберем запуск только что созданной функции. В случае с Kotlin Multiplatform нам нужно выбрать задачу под таргет (одну или все), в рамках которой мы запустим тест (рис. 9.6).



Рис. 9.6. Выбор задачи для запуска

Это позволит проверить корректность работы нашего commonMain под конкретный таргет.

Выберем задачу (например, для Android) и проверим запуск теста (рис. 9.7). Пока он не содержит никаких инструкций, поэтому выполнится успешно.

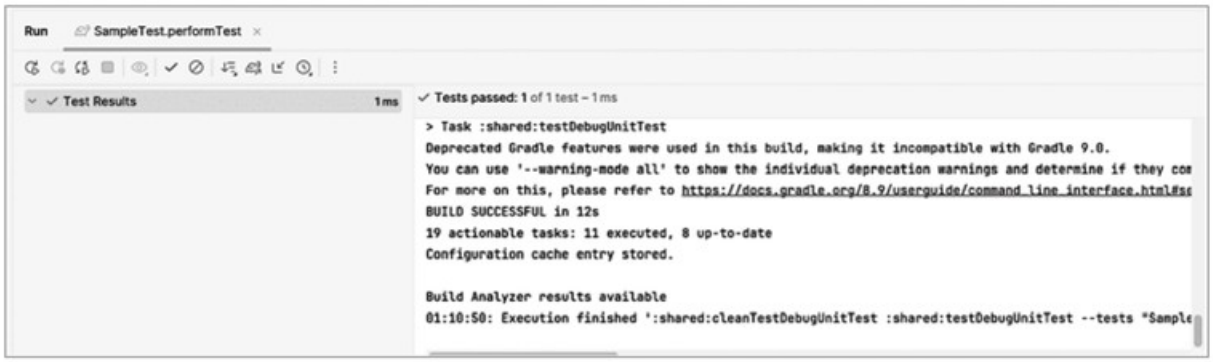


Рис. 9.7. Результат запуска пустого теста

Теперь напишем логику проверки. Для валидации результата используются специальные функции assert. Они подробно описаны в табл. 9.1.

Таблица 9.1

Assert	Действие
`fun assertTrue(condition: Boolean, message: String? = null)`	Проверяет, что условие true. Если условие false, тест завершается с ошибкой
`fun assertFalse(condition: Boolean, message: String? = null)`	Проверяет, что условие false. Если условие true, тест завершается с ошибкой
`fun assertEquals(expected: Any?, actual: Any?, message: String? = null)`	Проверяет, что два значения равны. Если значения не равны, тест завершается с ошибкой
`fun assertNotEquals(illegal: Any?, actual: Any?, message: String? = null)`	Проверяет, что два значения не равны. Если значения равны, тест завершается с ошибкой
`fun assertNull(actual: Any?, message: String? = null)`	Проверяет, что значение null. Если значение не null, тест завершается с ошибкой
`fun assertNotNull(actual: Any?, message: String? = null)`	Проверяет, что значение не null. Если значение null, тест завершается с ошибкой
`fun assertSame(expected: Any?, actual: Any?, message: String? = null)`	Проверяет, что два объекта ссылаются на один и тот же экземпляр. Если объекты разные, тест завершается с ошибкой
`fun assertNotSame(illegal: Any?, actual: Any?, message: String? = null)`	Проверяет, что два объекта не ссылаются на один и тот же экземпляр. Если объекты одинаковые, тест завершается с ошибкой
`fun assertFails(block: () -> Unit): Throwable`	Проверяет, что блок кода выбрасывает исключение. Если исключение не выбрасывается, тест завершается с ошибкой

Рассмотрим самый простой пример (листинг 9.3).

Листинг 9.3. Пример простого теста Kotlin

```
@Test
fun testSameObject() {
    val obj1 = "Kotlin"
    val obj2 = obj1
    assertSame(obj1, obj2)
}
```

Здесь у нас две переменные ссылаются на один объект со значением `Kotlin`. Функция `assertSame` вернет в качестве итога успешный результат.

Напишем теперь то, что будет для нас действительно полезно. Попробуем проверить десериализацию JSON с помощью `JsonDecoder` из главы 5 про сетевой слой (листинг 9.4).

Листинг 9.4. JsonDecoder для десериализации

```
class JsonDecoder{
  companion object {
    val instance = JsonDecoder()
    val jsonDecoder = kotlinx.serialization.json.Json { ignoreUnknownKeys = true
}
  }

  inline fun<reified T> decode(json:String): T {
    return jsonDecoder.decodeFromString(json)
  }
}
```

Добавим тестовые данные. Нам потребуется json-строка с сериализованным объектом `NewsItem`, возьмем ее из примера API news.api.org, и один объект с полями, заполненными теми же значениями (листинг 9.5).

Листинг 9.5. Тестовые данные

```
//Тестовый объект в виде JSON
val testNewsJSON = """
    {"author": "las vegas review-journal",
    "title": "TESLAS TORCHED IN VEGAS...",
    "description": "Police are investigating after an individual is believed to
have set several vehicles on fire at a Tesla service center in Las Vegas.",
    "url": "https://www.reviewjournal.com/post/3322726",
    "urlToImage": "https://www.reviewjournal.com/wp-content/
uploads/2025/03/20271272_web1_Teslafire_c2420f.jpg",
    "publishedAt": "2025-03-18T18:00:03Z",
    "content": "A suspect police who said set multiple Teslas on fire, used
Molotov cocktails and shot three rounds into the vehicles at a Tesla Collision
Center early Tuesday morning is still on the loose, accordin… [+1891 chars]"
    }
""".trimIndent()

// Поля объекта должны быть идентичны значениям в JSON testNewsJSON
```

```
val testNewsJsonObject = NewsItem(
        author = "las vegas review-journal",
        title = "TESLAS TORCHED IN VEGAS...",
        description = "Police are investigating after an individual is believed
to have set several vehicles on fire at a Tesla service center in Las Vegas.",
        url = "https://www.reviewjournal.com/post/3322726",
        urlToImage = "https://www.reviewjournal.com/wp-content/
uploads/2025/03/20271272_web1_Teslafire_c2420f.jpg",
        publishedAt = "2025-03-18T18:00:03Z",
        content = "A suspect police who said set multiple Teslas on fire, used
Molotov cocktails and shot three rounds into the vehicles at a Tesla Collision
Center early Tuesday morning is still on the loose, accordin… [+1891 chars]"
)
```

Теперь наполним логикой функцию `performTest`. Проверим корректность работы JsonDecoder с помощью `assertEquals` (листинг 9.6).

Листинг 9.6. Проверка работы декодера

```
@Test
fun performTest() {
    val newsItem = JsonDecoder.instance.decode<NewsItem>(testNewsJSON)
    assertEquals(testNewsJsonObject, newsItem)
}
```

В нашем случае тест завершится успешно. Выглядит хорошо на первый взгляд. Однако, по правилам, каждый тест должен проверять только один из случаев работы определенного объекта. Значит, если для полной проверки нужно будет описать несколько кейсов, каждый из них должен быть в своем методе с аннотацией `@Test`. Инициализация проверяемого объекта, а также всех его зависимостей в каждом методе теста будет неэффективной и затратной в плане ресурсов. Поэтому стоит задекларировать такие переменные в области видимости класса, а не функции, а их экземпляры создать и присвоить переменным или свойствам в специальном методе с аннотацией `@BeforeTest` (листинг 9.7).

Листинг 9.7. Инициализация инстансов в методе @BeforeTest

```
private lateinit var jsonDecoder: JsonDecoder

@BeforeTest
fun setup() {
  jsonDecoder = JsonDecoder.instance
}
```

Такой метод выполнится один раз при запуске тестов всего класса.

9.2. Работа с моками для комплексных тестов

Перейдем к кейсам посложнее и покроем тестами сервисный и сетевой слои. Обычно, чтобы упростить реализацию тестов, сократить и оптимизировать создание инстансов объектов-зависимостей либо получить ответ на некоторые

запросы, используют генераторы моков (Mocks). Популярными решениями являются: Mokks, Mokkery, Mockito, MockingBird, Mockative, MocKMP и другие.

Нам нужна библиотека, которая точно поддерживает Kotlin Multiplatform, поэтому добавим в наш проект Mokkery https://mokkery.dev/ (рис. 9.8).

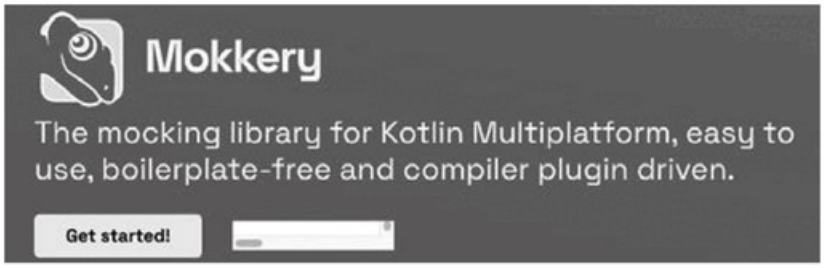


Рис. 9.8. Решение Mokkery

Библиотека поддерживает работу как с commonMain, так и со всеми нужными нам таргетами:

- JVM/Android;
- Native (iOS, Desktop Linux, Mac OS);
- WASM;
- JS.

Для установки потребуется добавить плагин в секцию плагинов `build.gradle.kts` shared-модуля (листинг 9.8).

Листинг 9.8. Подключение mokkery как плагина

```
plugins {
    kotlin("multiplatform")
    id("dev.mokkery") version "2.7.1"
}
```

Создадим тест для проверки логики NewsService (листинг 9.9).

Листинг 9.9. Тестирование NewsService. Моки инстансов

```
class NewsServiceTest {

  private lateinit var newsService: NewsService
  private lateinit var networkClient: NetworkClient

  @BeforeTest
  fun setup() {
    // 1 Инициализируем инстансы
    networkClient = mock<NetworkClient>() // Используем мок зависимостей
                                          для тестов
    newsService = NewsService(networkClient)
  }
}
```

Нам потребуется инициализировать инстанс `NewsService` и его зависимости `NetworkClient`. Для зависимостей мы будем создавать экземпляры сервисов через инструкцию `mock`. Делаем это в методе `setup`, помеченном аннотацией `@BeforeTest`.

Сетевые запросы в нашем коде идут асинхронно, с использованием корутин. Чтобы мы могли протестировать корректно описанные `suspend`-методы, нам нужно добавить специальную версию библиотеки корутин для тестов (листинг 9.10).

Листинг 9.10. Подключение библиотеки корутин для тестирования

```
commonTest.dependencies {
  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.10.1")
}
```

Теперь нужно проинициализировать и подключить специальный тестовый диспетчер корутин (листинг 9.11).

Листинг 9.11. Подключение тестового диспетчера корутин

```
class NewsServiceTest {

  private val testDispatcher = StandardTestDispatcher()
  //...

  @BeforeTest
  fun setup() {
    //....
    // 2 Создаем тестовый диспетчер
    Dispatchers.setMain(testDispatcher)
  }
}
```

Создадим тестовый диспетчер типа `StandardTestDispatcher` и подключим его через `Dispatchers.setMain` как диспетчер главного потока.

Также нужно не забыть отключить тестовый диспетчер по завершении всех тестов (листинг 9.12).

Листинг 9.12. Отключение тестового диспетчера

```
@AfterTest
fun tearDown() {
     // Отключаем диспетчер после завершения теста
     Dispatchers.resetMain()
}
```

Для этого в методе с аннотацией `@AfterTest` нужно вызвать `Dispatchers.resetMain`.

Мы будем тестировать метод `loadNews`, поэтому нужны тестовые данные для имитации ответа `Result<NewsList>` (листинг 9.13).

Листинг 9.13. Тестовые данные

```
val testNewsJson = NewsList(
            articles = listOf(
                    NewsItem(
                            author = "las vegas review-journal",
                            title = "TESLAS TORCHED IN VEGAS...",
                            description = "Police are investigating after an
individual is believed to have set several vehicles on fire at a Tesla service
center in Las Vegas.",
                            url = "https://www.reviewjournal.com/post/3322726",
                            urlToImage = "https://www.reviewjournal.com/wp-
content/uploads/2025/03/20271272_web1_Teslafire_c2420f.jpg",
                            publishedAt = "2025-03-18T18:00:03Z",
                            content = "A suspect police who said set multiple
Teslas on fire, used Molotov cocktails and shot three rounds into the vehicles
at a Tesla Collision Center early Tuesday morning is still on the loose,
accordin… [+1891 chars]"
                    )
            )
    )
```

Почти то же самое, что в предыдущем тесте, только теперь будем использовать другую модель данных.

В итоге у нас получится такой тест-кейс (листинг 9.14).

Листинг 9.14. Тест запроса новостей

```
@Test
    fun `fetch news successfully`() = runTest {
        // 3 Ответ на метод loadNews
        val result = Result.success(testNewsJson)

        // 4 Создаем мок-запрос для сетевого клиента. При любых входных отдаем
           наш result
        everySuspend<Result<NewsList>> { networkClient.request(any()) }
            returns result

        // 5 Вызываем метод, который нужно протестировать
        verifySuspend {
            newsService.loadNews()
        }
    }
```

Давайте разберем его детально. Мы тестируем запрос `loadNews` сервиса `NewsService`, поэтому необходимо замокировать поведение его зависимости `networkClient` и его метода `request`. Для этого используем метод `everySuspend` (листинг 9.15).

Листинг 9.15. Блок everySuspend

```
everySuspend<Result<NewsList>> { networkClient.request(any()) }
            returns result
```

Блок `every{}` позволяет задать поведение метода или свойства инстанса типа при определенных или любых условиях. Если наш метод является `suspend`, то тогда для мока его вызова используется `everySuspend`.

Порядок вызовов в тесте мы можем проверить с помощью блоков `verify` и `verifySuspend` для вызова соответствующих корутин.

Запустим написанный тест и получим ошибку (листинг 9.16).

Листинг 9.16. Ошибка запуска теста

```
type 'network.NetworkClient' is final and cannot be used with 'mock'!
```

Мы не можем использовать мок для генерации инстанса финального класса. Поставим модификатор open для `NetworkClient`. Запустим тест снова и получим теперь новую ошибку (листинг 9.17).

Листинг 9.17. Ошибки запуска теста с финальными членами

```
Type 'network.NetworkClient' has final members and cannot be used with 'mock'!
Final members: suspend fun <reified T> request(path: String): Result<T>, suspend
fun <reified T> request(path: String, method: Method, body: Any? = ...):
Result<T>, val httpClient: HttpClient

`everySuspend`/`verifySuspend` does not support actual suspension! Only mock
method calls are allowed!
dev.mokkery.internal.IllegalSuspensionException: `everySuspend`/`verifySuspend`
does not support actual suspension! Only mock method calls are allowed!
      at app//dev.mokkery.internal.utils.RunSuspensionKt.runSuspension
              (RunSuspension.kt:11)
      at app//dev.mokkery.internal.EveryKt.internalEverySuspend$lambda$0
              (Every.kt:17)
      at app//dev.mokkery.internal.EveryKt.internalEvery(Every.kt:23)
      at app//dev.mokkery.internal.EveryKt.internalEverySuspend(Every.kt:17)
      at app//NewsServiceTest$fetch news successfully$1.invokeSuspend
             (NewsServiceTest.kt:56)
      at app//NewsServiceTest$fetch news successfully$1.invoke
             (NewsServiceTest.kt)
      at app//NewsServiceTest$fetch news successfully$1.invoke
              (NewsServiceTest.kt)
```

Проблема в том, что наши методы запроса не являются открытыми для переопределения и виртуализации (не open). Но добавить модификатор open просто так у нас сейчас не получится. Мы реализовали методы класса как обобщения с reified inline, а такие функции не могут быть открытыми (open) (рис. 9.9).



Рис. 9.9. Проблемы с inline и open

Для получения и распознавания тела ответа (body) из результата запроса у нас используется специальный метод, который тоже реализован как обобщение reified inline. И нам придется от него избавиться (листинг 9.18).

Листинг 9.18. Generic-метод запроса и десериализации

```
suspend inline fun <reified T> request(path: String): Result<T> {
            return try {
                val data = httpClient.get(path)
               val result = data.body<T>()
                Result.success(result)
            } catch (e: Exception) {
                Result.failure(e)
            }
        }

//Метод body<T> сам reified inline
public suspend inline fun <reified T> HttpResponse.body(): T = call.
bodyNullable(typeInfo<T>()) as T
```

Попробуем решить задачу по-другому. Создадим еще один сетевой клиент и реализуем в нем метод запроса без использования модификатора reified inline (листинг 9.19).

Листинг 9.19. Тестовый сетевой клиент

```
open class OpenNetworkClient () {
  open val httpClient = createHttpClient()

  @OptIn(InternalAPI::class)
  open suspend fun<T:Any> request(path: String,
    deserializationStrategy: DeserializationStrategy<T>): Result<T> {
    return try {
      val data = httpClient.get(path)
      print(data)
      // Получаем строку
      val test = data.bodyAsText()
      // Десериализируем с явной передачей сериализатора типа
      val result = JsonDecoder.instance
        .decodeString(deserializationStrategy,test) as T
      Result.success(result)
    } catch (e: Exception) {
      Result.failure(e)
    }
  }
```

Мы переписали метод `request` для поддержки работы с любыми типами, которые наследуют `Any`. Потребовалось добавить также специальную версию декодера с явной передачей сериализатора типа (листинг 9.20).

Листинг 9.20. Доработка декодера

```
class JsonDecoder{

  companion object {
```

```
    val instance = JsonDecoder()
    val jsonDecoder = kotlinx.
      serialization.json.
      Json { ignoreUnknownKeys = true }
  }

  fun<T:Any> decodeString(
    deserializationStrategy: DeserializationStrategy<T>,
    json:String): T? {

// Передаем десериализатор явно
    return jsonDecoder.
      decodeFromString(deserializationStrategy, json)
  }
}
```

Вызов такого метода из `NewsService` будет выглядеть так (листинг 9.21):

Листинг 9.21. Вызов метода с передачей десериализатора

```
class OpenNewsService(private val httpClient: OpenNetworkClient) {

  open suspend fun loadNews(): Result<NewsList> {
    return httpClient.request(URL, NewsList.serializer())
  }
  //...
}
```

В тестовых целях мы заменили сетевой клиент не в исходном сервисе, а в другом классе, тоже созданном для теста.

Попробуем использовать для теста `NewsServiceTest` новые типы (листинг 9.22).

Листинг 9.22. Исправленная версия теста

```
class NewsServiceTest {

  //...

// Новые типы для сервиса и клиента
  private lateinit var newsService: OpenNewsService
  private lateinit var networkClient: OpenNetworkClient

  //...

  @Test
  fun `fetch news successfully`() = runTest {
    // 3 Ответ на метод loadNews
    val result = Result.success(testNewsJson)
    // 4 Создаем мок-запрос для сетевого клиента. При любых входных отдаем
       наш result
    everySuspend<Result<NewsList>> { networkClient.request(any(), any()) }
returns result
```

```
    // 5 Вызываем метод, который нужно протестировать
    newsService.loadNews()

    verifySuspend {
    networkClient.request(URL, NewsList.serializer())
    }

  }
}
```

Обратите внимание: сигнатура вызова `networkClient.request` тоже изменилась.

Запустим тест и проверим (рис. 9.10).

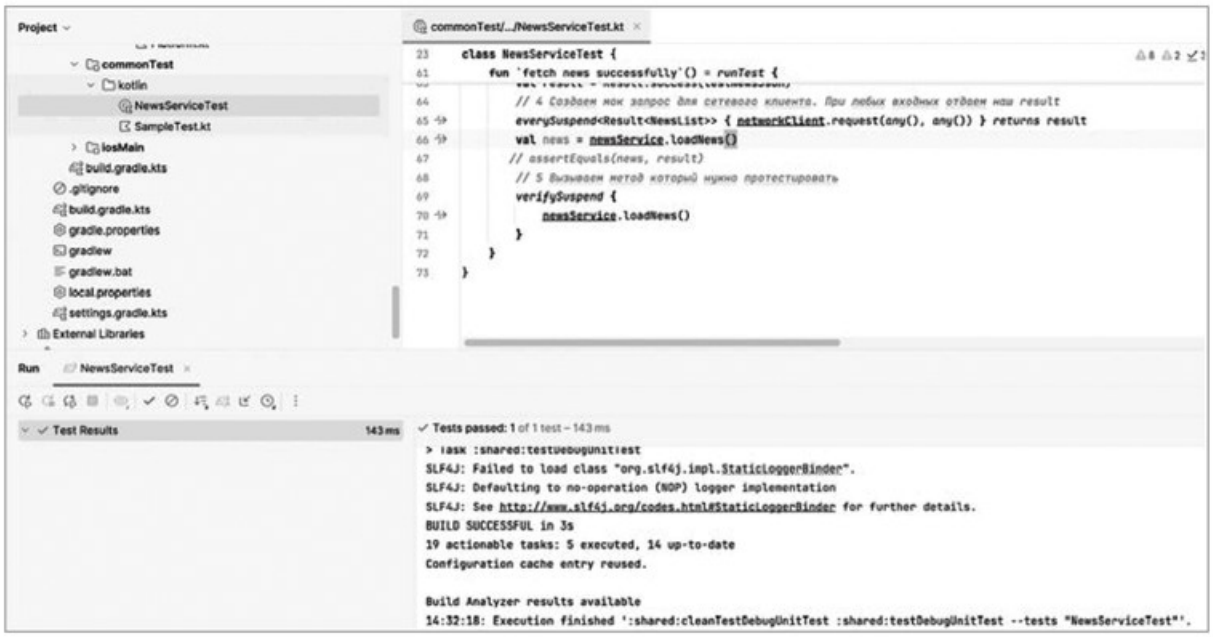


Рис. 9.10. Готовый тест на Kotlin Test

Промежуточные выводы

Для того чтобы поддержать покрытие тестом, нам пришлось переписать часть функционала. Это означает, что для реализации настоящего приложения такие нюансы реализации стоит продумывать заранее. Также имеет смысл рассмотреть другое решение для тестирования кода.

9.3. Платформенные тесты

Мы можем использовать Kotlin Test и для платформенных тестов. Это позволит протестировать Android- и iOS-таргеты и проверить не только специфичный для платформы функционал, но и кастомизированное поведение.

В качестве примера рассмотрим покрытие тестами хранилища Android `SharedPreferences` и хранилища iOS `UserDefaults`.

Для локальных тестов Android создадим специальный таргет. Выберем `androidInstrumentedTest`, чтобы поддержать работу со специфическим API Android SDK (рис. 9.11).



Рис. 9.11. Тестовый Android-таргет

Добавим описание таргета в `build.gradle.kts` shared-модуля. Также добавим библиотеки `junit` для нативного тестирования Android и core для поддержки доступа к возможностям SDK (листинг 9.23).

Листинг 9.23. Настройка тестового Android-таргета

```
androidInstrumentedTest.dependencies {
  implementation(libs.kotlin.test)
  implementation(libs.androidx.core.ktx)
  implementation(libs.androidx.junit.ktx)
}

//
androidx-core-ktx = { group = "androidx.test", name = "core-ktx",
                      version.ref = "1.6.1" }
androidx-junit-ktx = { group = "androidx.test.ext", name = "junit-ktx",
                       version.ref = "1.2.1" }
```

Запустим синхронизацию Gradle и приступим к написанию теста.

Ознакомьтесь с тестом в листинге 9.24. Основное его отличие от примеров, которые мы рассматривали выше, состоит в использовании `@RunWith` с указанием AndroidJUnit4. Эта аннотация нужна для запуска инструментального теста Android на движке AndroidJUnit4. Так мы сможем использовать в нашем тесте для Android SDK.

Листинг 9.24. Тест Android

```
@RunWith(AndroidJUnit4::class)
class SharedPreferencesTest {

  private lateinit var sharedPreferences: SharedPreferences
  private val expectedAppStartCount = 10

  @BeforeTest
  fun setup() {
    // Инициализируем контекст и sharedPreferences
    val context = ApplicationProvider.getApplicationContext<Context>()
    sharedPreferences = context.getSharedPreferences(SHARED_PREF_NAME,
    Context.MODE_PRIVATE)
  }
```

```
  @AfterTest
  fun tearDown() {
    // Зачищаем sharedPreference
    sharedPreferences.edit().clear().apply()
  }

  @Test
  fun checkAppStartCountStoresCorrectly() {
    //  Сохраняем значение
    sharedPreferences.edit().putInt(KEY_APP_START_COUNT,
    expectedAppStartCount).apply()

    val actualAppStartCount = sharedPreferences.getInt(KEY_APP_START_COUNT, -1)
    // Проверяем
    assertEquals(expectedAppStartCount, actualAppStartCount)
  }

  companion object {
    private const val SHARED_PREF_NAME = "test_prefs"
    private const val KEY_APP_START_COUNT = "app_start_count"
  }
}
```

Пройдемся по функционалу этого теста:

1. В методе `@BeforeTest` мы запрашиваем контекст приложения `ApplicationContext`. Через контекст мы запрашиваем доступ к внутреннему хранилищу на `SharedPreferences`.
2. В методе `@AfterTest` нужно вызвать зачистку используемого внутреннего хранилища.
3. В самом тесте мы сначала сохраняем тестовое значение по ключу, затем запрашиваем его по этому же ключу и сравниваем исходное значение с полученным результатом запроса.

Все довольно просто. Повторим то же самое с iOS. Для тестирования `UserDefaults` не потребуется подключать какие-либо дополнительные зависимости, это сугубо нативный функционал iOS SDK.

Начнем с создания таргета (рис. 9.12).

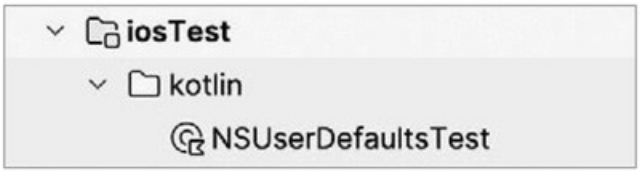


Рис. 9.12. Тестовый таргет для iOS

Пропишем таргет в `build.gradle.kts` и добавим зависимость от Kotlin Test (листинг 9.25).

Листинг 9.25. Настройка таргета iOSTest

```
iosTest.dependencies {
  implementation(libs.kotlin.test)
}
```

Теперь напишем тест для хранилища UserDefaults (листинг 9.26).

Листинг 9.26. Тест для UserDefaults

```
class NSUserDefaultsTest {

  private lateinit var userDefaults: NSUserDefaults
  private val expectedAppStartCount = 10

  @BeforeTest
  fun setup() {
    // Создаем инстанс, который будем использовать для тестов
    userDefaults = NSUserDefaults.standardUserDefaults()
  }

  @AfterTest
  fun teardown() {
    // После теста удаляем данные из хранилища
    userDefaults.removeObjectForKey(KEY_APP_START_COUNT)
  }

  @Test
  fun `test app start count stores correctly in NSUserDefaults`() {
    userDefaults.setObject(expectedAppStartCount, forKey = KEY_APP_START_COUNT)
    val actualAppStartCount = userDefaults.integerForKey(KEY_APP_START_COUNT)
    assertEquals(expectedAppStartCount, actualAppStartCount.toInt())
  }

  companion object {
    private const val KEY_APP_START_COUNT = "app_start_count"
  }
}
```

По функционалу тест идентичен тесту для Android. Единственное отличие — мы используем `NSUserDefaults` вместо `SharedPreferences`.

9.4. Expect/actual в тестах

Еще одним вариантом реализации платформенных тестов является использование `expect/actual` для поддержки разного поведения тестов. Например, мы можем создать декларацию для общего теста хранилища (листинг 9.27).

Листинг 9.27. Пример общей декларации теста

```
expect class SampleStorageTest {

    @BeforeTest
```

```
    fun setup()

    @AfterTest
    fun teardown()

    @Test
    fun testStorage()
}
```

Как мы помним из главы 3 про общее и нативное, все эти методы являются обязательными к `actual`-реализации.

Модифицируем код iOS, как в листинге 9.28.

Листинг 9.28. Actual-реализация на iOS

```
actual class SampleStorageTest {

  private lateinit var userDefaults: NSUserDefaults
  private val expectedAppStartCount = 10

  @BeforeTest
  actual fun setup() {
    // Создаем инстанс, который будем использовать для тестов
    userDefaults = NSUserDefaults.standardUserDefaults()
  }

  @AfterTest
  actual fun teardown() {
    // После теста удаляем данные из хранилища, чтобы не влияли на другие тесты
    userDefaults.removeObjectForKey(KEY_APP_START_COUNT)
  }

  @Test
  actual fun testStorage() {
    userDefaults.setObject(expectedAppStartCount, forKey = KEY_APP_START_COUNT)
    val actualAppStartCount = userDefaults.integerForKey(KEY_APP_START_COUNT)
    assertEquals(expectedAppStartCount, actualAppStartCount.toInt())
  }

  companion object {
    private const val KEY_APP_START_COUNT = "app_start_count"
  }
}
```

Мы взяли предыдущую реализацию и перенесли код в методы, помеченные `actual`.

Но когда мы попробуем повторить аналогичные действия для Android, то обнаружим, что наши `actual`-реализации не видят деклараций `expect`. Дело в том, что мы неверно задали тестовый таргет. Он подойдет для проверки кода Android-приложения, но чтобы тесты Android видели декларации `commonTest`, нужно создать таргет `AndroidUnitTest` и перенести их туда (листинг 9.29).

Листинг 9.29. Новый таргет для тестов Android

```
androidUnitTest.dependencies {
                    implementation(libs.kotlin.test)
                    implementation(libs.androidx.core.ktx)
                    implementation(libs.androidx.junit.ktx)
                }
```

Тест увидит `expect`-декларации, но при запуске мы получим другую проблему (листинг 9.30).

Листинг 9.30. Фрагмент стектрейса

```
No instrumentation registered! Must run under a registering instrumentation.
java.lang.IllegalStateException: No instrumentation registered! Must run under
          a registering instrumentation.
  at androidx.test.platform.app.InstrumentationRegistry.getInstrumentation
          (InstrumentationRegistry.java:45)
  at androidx.test.core.app.ApplicationProvider.getApplicationContext
          (ApplicationProvider.java:41)
  at ru.azharkova.sampletest.SampleStorageTest.setup(SampleStorageTest.kt:19)
  at java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Unknown
          Source)
  at java.base/java.lang.reflect.Method.invoke(Unknown Source)
```

Дело в том, что в этом таргете недоступен инструментарий, который дает доступ к тестированию компонентов Android SDK. Добавление аннотации и `@RunWith(AndroidJUnit4::class)` и всех нужных библиотек тоже не позволит корректно запустить такой тест.

При работе с тестами под Android и другими платформенными таргетами действуют следующие правила:

1. Желательно покрывать логику общего модуля в `commonTest`. Запуск под тем или иным таргетом обеспечит то, что под капотом будет запрошена версия кода именно под выбранную платформу.
2. Используйте `expect/actual` только в крайнем случае.
3. Реализация тестов для Android-таргета возможна либо в `androidUnitTest`, либо в `androidInstrumentedTest`. Используйте `androidUnitTest` для переопределения кода из commonTest, а `androidInstrumentedTest` подойдет, если нужно тестировать код, связанный с AndroidSDK. Но `androidInstrumentedTest` не видит `commonTest`.

Мы не стали рассматривать, как подключать нативные компоненты тестирования на Android и iOS, потому что их не получится никаким образом соединить в общей части, и это обеспечит тестовое покрытие только того функционала, который не является общим. Сейчас мы не планируем покрывать тестами UI.

9.5. Kotest

Рассмотрим еще одно решение для реализации тестов и покрытия ими кода в приложении Kotlin Multiplatform. Это кросс-платформенная версия библиотеки Kotest (https://kotest.io/), популярной среди Android-разработчиков. У нее довольно широкий функционал, включая встроенные средства проверки. Библиотека позволяет писать property-based, модульные и интеграционные тесты. Также она легко интегрируется с другими решениями (Koin, Spring, Mockk, Mockkery и т. п.), поддерживает многопоточность и многое другое.

Для подключения решения к нашему проекту добавим следующие зависимости (листинг 9.31).

Листинг 9.31. Настройка зависимости Kotest

```
commonTest.dependencies {
  implementation(libs.kotest.assertions.core)
  implementation(libs.kotest.framework.engine)
}

// Gradle libs
kotest = "6.0.0.M1"

[libraries]
kotest-assertions-core = { module = "io.kotest:kotest-assertions-core",
                           version.ref = "kotest" }
kotest-framework-engine = { module = "io.kotest:kotest-framework-engine",
                            version.ref = "kotest" }
kotest-runner-junit5 = { module = "io.kotest:kotest-runner-junit5",
                         version.ref = "kotest" }
```

Обратите внимание, что не все версии Kotest поддерживают Kotlin Multiplatform.

Для работы с Android-таргетом нам обязательно нужно установить движок Junit5, иначе мы не сможем запустить тесты. Также поддержку запуска тестов на движке нужно прописать в настройках таргета в `build.gradle.kts` (листинг 9.32).

Листинг 9.32. Настройка подключения к таргету в build.gradle.kts

```
androidUnitTest.dependencies {
    implementation(libs.kotest.runner.junit5)
//...
}

// Настройка
androidTarget {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions {
                    jvmTarget.set(JvmTarget.JVM_1_8)
                }
```

```
            }
        }
    }
    android.testOptions {
        unitTests.all {
            it.useJUnitPlatform()
        }
    }
```

Приступим к написанию теста. Покроем класс `NewsService`, но уже с помощью `Kotest` (листинг 9.33).

Листинг 9.33. Тест для NewsService на Kotest

```
@OptIn(kotlinx.coroutines.ExperimentalCoroutinesApi::class)
class NewsClientTest : BehaviorSpec({

  val mockNetworkClient = mock<OpenNetworkClient>()
  val newsClient = OpenNewsService(mockNetworkClient)

  Given("should check load news") {
    val result = Result.success(testNewsJson)
    everySuspend<Result<NewsList>> { mockNetworkClient.request(any(), any()) }
returns result

    When("fetching news") {
      newsClient.loadNews()
      Then("should return news list") {
        verifySuspend {
          mockNetworkClient.request(URL, NewsList.serializer())
        }
      }
    }
  }
})
```

Давайте разберем код этого теста.

В Kotest тестовые методы группируются по специальным классам-наследникам класса `Spec` (спецификация теста). Спецификация позволяет как задать вид и стиль теста, так и определить его назначения. Например, `FunSpec` часто используется для тестирования отдельных функций (листинг 9.34).

Листинг 9.34. Пример теста FunSpec

```
class StringCalculatorTest : FunSpec({

    test("empty string results in zero") {
        add("") shouldBe 0
    }
})
```

Наша цель — проверить поведение компонента в самых разных условиях. Поэтому нам подойдет тест в формате `BehaviorSpec` с использованием подхода BDD

(behave-driven-development). По этой спецификации мы структурируем тест по блокам `Given-When-Then`. В `Given` мы задаем стартовые условия реализации описываемого тест-кейса, в `When` вызываем событие-запуск, в `Then` проверяем либо результат выполнения с помощью функций `assert` или `shouldBe`, либо последовательность действий.

В нашем случае в блоке `Given` мы зададим поведение сетевого клиента и его ответ на запрос `request` (листинг 9.35).

Листинг 9.35. Блоки Given-When-Then

```
Given("should check load news") {

// 1 Задаем поведение mockNetworkClient
    val result = Result.success(testNewsJson)
    everySuspend<Result<NewsList>> { mockNetworkClient.request(any(), any()) }
returns result

     When("fetching news") {

// 2 Вызываем триггерное событие
      newsClient.loadNews()

      Then("should return news list") {

// 3 Проверяем последовательность
        verifySuspend {
          mockNetworkClient.request(URL, NewsList.serializer())
        }
      }
    }
```

В блоке `When` вызовем триггер проверяемого события `newsClient.loadNews`. Затем в блоке `Then` проверяется последовательность вызовов.

Обратите внимание, что для моков и генерации заглушек событий и инстансов мы используем то же решение на Mokkery, что и для Kotlin Test.

Теперь запустим тест и проверим результат. ДляAndroid-таргета тест запускается стандартно, через контекстное меню запуска тестовых задач (рис. 9.13).

Для нативных таргетов (iOS, Linux, MacOS, native) рекомендуется запускать тесты через команду терминала:

```
./gradlew check
```

С результатами тестов можно будет ознакомиться в сгенерированном в формате html отчете (рис. 9.14).

Если вы мигрируете с нативного Android-приложения на кросс-платформенное и у вас уже реализовано покрытие бизнес-логики тестами на Kotest, то, возможно, вам будет проще перенести их и адаптировать под commonTest.



Рис. 9.13. Запуск теста Kotest для Android

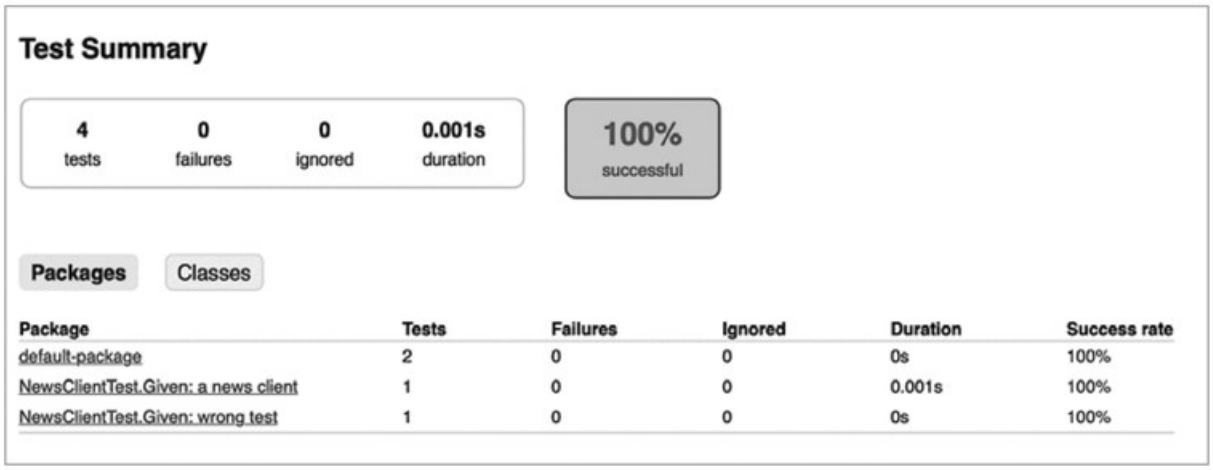


Рис. 9.14. Результаты запуска Kotest-теста с помощью check

ПОДВЕДЕМ ИТОГИ

- ✓ Kotlin Test — удобное и гибкое решение для тестирования кросс-платформенных приложений.
- ✓ В дополнение к нему можно использовать библиотеки для моков, а также библиотеки для тестирования корутин.
- ✓ В качестве альтернативного решения можно использовать библиотеку Kotest.

9.6. Применение на практике

Мы реализовали тест для `NewsService`, и даже в нескольких вариантах, а также создали тест для JsonDecoder. Добавим покрытие на оставшиеся элементы бизнес-логики:

1. Сетевой клиент `NetworkClient` содержит логику сторонней библиотеки Ktor. На этом уровне мы можем реализовать покрытие только того, что мы написали сами. Тест для JsonDecoder у нас уже есть (см. листинги 9.3–9.6).

   Android Studio поддерживает генерацию каркаса тестов на разных движках, включая Kotest. Для автоматического создания теста выбираем нужный класс, в контекстном меню выбираем Create test и задаем нужные настройки (рис. 9.15).

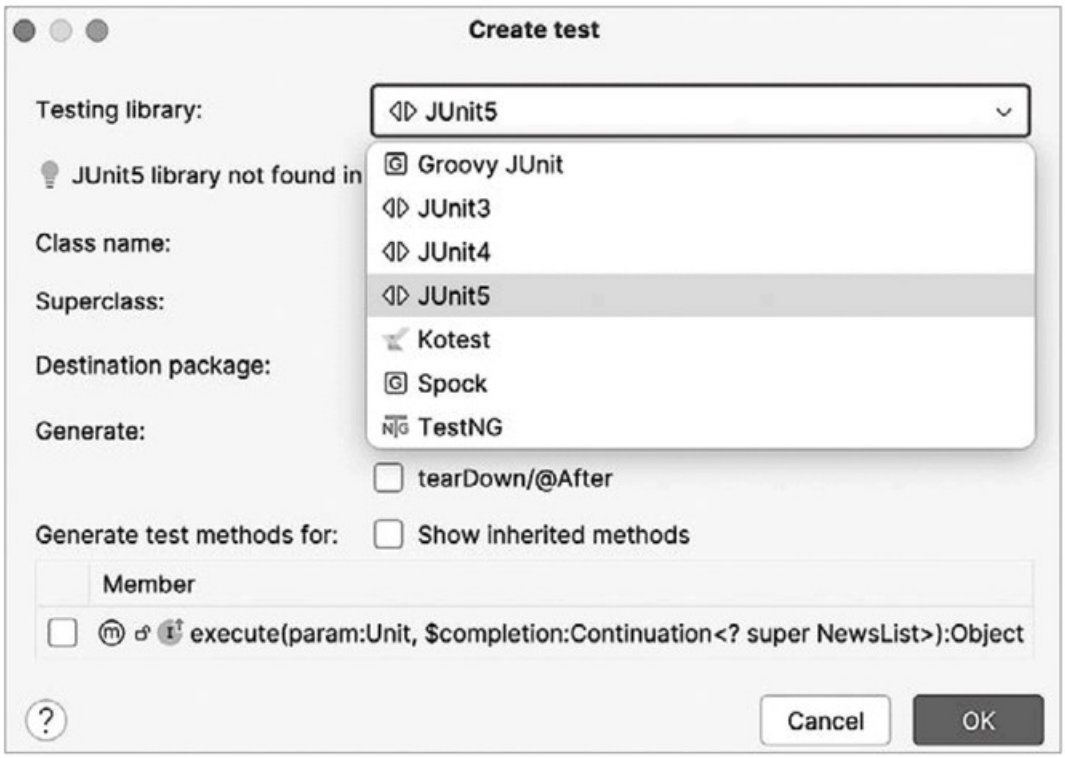


Рис. 9.15. Автоматическое создание теста

2. Создадим каркасы для `ViewModel` и `UseCase`.

   Обратите внимание, что будет создан тест в формате `FunSpec`. Такое поведение нас вполне устроит, потому что в этих компонентах мы будем тестировать методы, которые выполняют одну операцию. При добавлении тест автоматически попадет в тестовый таргет под Android, и его придется перенести в `commonTest`.

3. Напишем тест для `UseCase`. Нам потребуется внести изменения в код, чтобы классы наших зависимостей были open, как и их неприватные методы.

   В листинге 9.36 приведен пример реализации теста.

Листинг 9.36. Реализация UseCase теста

```
class NewsUseCaseTest : FunSpec({

  val newsService = mock<OpenNewsService> {
```

```
        everySuspend { loadNews() } returns Result.success(testNewsJson)
    }
    val newsUseCase = NewsUseTestCase(newsService)

    test("NewsUseCase execute") {
        val result = newsUseCase.execute(Unit)
        result shouldBe testNewsJson
    }
})
```

Мы мокируем и `NewsService`, и его поведение. В блоке `test` мы вызовем метод execute инстанса используемого `UseCase` и сравним с эталонным результатом.

4. То же самое делаем для `ViewModel`. Создаем каркас теста, переносим его в commonTest и наполняем кодом (листинг 9.37).

Листинг 9.37. Реализация теста для ViewModel

```
class NewsViewTestModelTest : FunSpec({

    val useCase = mock<NewsUseTestCase> {
        everySuspend {
            execute(Unit)
        } returns testNewsJson
        everySuspend {
            invoke(Unit)
        } returns Result.success(testNewsJson)
    }

    test("check loadNews") {
        val result = useCase.invoke(Unit)
        result shouldBe Result.success(testNewsJson)
    }
})
```

Готово. Мы покрыли тестами наше приложение и практически закончили с бизнес-логикой.

В следующей главе займемся реализацией общего UI нашего учебного приложения и навигацией.

ГЛАВА 10

Общий UI в приложениях Kotlin Multiplatform

В этой главе рассмотрим:

- концепцию общего UI в приложениях Kotlin Multiplatform;
- Compose Multiplatform;
- общую навигацию в UI, конфигурацию экранных модулей;
- нативный и общий UI;
- кросс-платформенный Server Driven UI;
- альтернативные решения UI для KMP.

10.1. Концепция общего UI

Чтобы понять концепцию и особенности работы с общим UI-слоем в кросс-платформенном приложении, стоит начать с изучения реализации UI в нативных приложениях. Изначально предполагалось, что в Kotlin Multiplatform UI будет реализовываться исключительно нативными средствами в нативных приложениях.

10.1.1. Нативный UI в iOS

Начнем с iOS. Стандартными способами работы с UI на этой платформе являются:

- верстка на Xib и сторибордах;
- верстка с помощью кода и нативных библиотек;
- декларативный фреймворк SwiftUI.

Долгое время основным подходом для создания UI и его верстки было использование Autolayout и Interface Builder. Interface Builder — это визуальный инструмент для проектирования пользовательского интерфейса в приложениях для iOS и macOS. Он входит в интегрированную среду разработки Xcode от Apple

и позволяет интерактивно в режиме реального времени работать с визуальными представлениями для отдельных `UIView` в виде Xib или `UIViewController` (рис. 10.1).



Рис. 10.1. Пример Storyboard в Interface Builder Xcode

При работе со Storyboards можно не только верстать UI для каждого `UIViewController`, но и настраивать навигацию и переходы между экранами по событиям на элементах UI, компонуя таким образом целую сцену или раздел.

Основным недостатком Interface Builder, Xib и Storyboards является сложность повторного использования сверстанных `UIViewController` и `UIView`, например, когда вам нужно использовать один и тот же экран в разных разделах. Также требуется время на вхождение в работу с Interface Builder.

Некоторые настройки UI через InterfaceBuilder также могут вызывать трудности: например, управление якорями (`NSContstraints`) для настройки положения между элементами, сокрытие и показ одних контролов и смещение других (листинг 10.1).

Листинг 10.1. Пример работы с NSConstraints в InterfaceBuilder

```
class TwoViewsViewController: UIViewController {

  // MARK: - IBOutlets для view
  @IBOutlet weak var firstView: UIView!
  @IBOutlet weak var secondView: UIView!

  // MARK: - IBOutlets для констрейнтов
  @IBOutlet weak var firstViewTopConstraint: NSLayoutConstraint!
  @IBOutlet weak var firstViewHeightConstraint: NSLayoutConstraint!
  @IBOutlet weak var secondViewTopConstraint: NSLayoutConstraint!
  @IBOutlet weak var secondViewHeightConstraint: NSLayoutConstraint!

  override func viewDidLoad() {
    super.viewDidLoad()
  }

  // MARK: - Примеры изменения констрейнтов

  // 1. Анимация изменения отступа верхней view
  func animateFirstViewTopConstraint() {
    UIView.animate(withDuration: 0.3) {
      self.firstViewTopConstraint.constant = 50  // Новое значение
      self.view.layoutIfNeeded()  // Обязательно для анимации констрейнтов
    }
  }

  // 2. Изменение высоты второй view
  func updateSecondViewHeight() {
    secondViewHeightConstraint.constant = 200
    // Без анимации:
    view.layoutIfNeeded()
  }
}
```

Альтернативный подход — создание верстки прямо в коде с помощью фреймов, якорей и решений типа SnapKit (листинг 10.2).

Листинг 10.2. Пример верстки с помощью библиотеки SnapKit

```
let box = UIView()
let container = UIView()

container.addSubview(box)

box.snp.makeConstraints { (make) -> Void in
    make.size.equalTo(50)
    make.center.equalTo(container)
}
```

Этот фрагмент кода позволяет создать UIView, добавить его в родительский контейнер, задать размер сторон, равный 50, и расположить по центру родителя.

Верстка кодом является более гибким подходом, решающим проблему с переиспользованием описанных элементов UI. Но код инициализации UI для визуальной части может быть очень громоздким. Работа с констрейнтами для настройки и/или изменения положения компонентов по отношению друг к другу тоже может быть непростой, а код — внешне избыточным (листинг 10.3).

Листинг 10.3. Пример работы с якорями (NSLayoutContstraints) с помощью кода

```
class TwoViewsViewController: UIViewController {

  let firstView = UIView()
  let secondView = UIView()

  override func viewDidLoad() {
    super.viewDidLoad()

    firstView.translatesAutoresizingMaskIntoConstraints = false
    secondView.translatesAutoresizingMaskIntoConstraints = false
    //...
    setupConstraints()
  }

  private func setupConstraints() {
    // Вариант 1: Одна view над другой с отступами

    NSLayoutConstraint.activate([
      // Первая view (верхняя)
      firstView.topAnchor.constraint(equalTo: view.safeAreaLayoutGuide.
                                     topAnchor, constant: 20),
      firstView.leadingAnchor.constraint(equalTo: view.leadingAnchor,
                                         constant: 20),
      firstView.trailingAnchor.constraint(equalTo: view.trailingAnchor,
                                          constant: -20),
      firstView.heightAnchor.constraint(equalToConstant: 150),

      // Вторая view (под первой)
      secondView.topAnchor.constraint(equalTo: firstView.bottomAnchor,
                                      constant: 20),
      secondView.leadingAnchor.constraint(equalTo: view.leadingAnchor,
                                          constant: 20),
      secondView.trailingAnchor.constraint(equalTo: view.trailingAnchor,
                                           constant: -20),
      secondView.heightAnchor.constraint(equalTo: firstView.heightAnchor),
    ])
  }
}
```

В 2019 году компания Apple представила на ежегодной сессии WWDC инновационный фреймворк — декларативный SwiftUI для быстрого и простого создания UI с помощью DSL-синтаксиса. Верстка, изменение UI, стилизация, навигация — все задается с помощью декларативного кода (листинг 10.4).

Листинг 10.4. Пример UI на SwiftUI

```
struct NewsListView: View {

    @State var data: [NewsItemsMock]

    init() {}

    var body: some View {
        NavigationView {
            List(data) { item in
                NavigationLink(destination: NewsItemView(item: item)) {
                    NewsItemRow(data: item)
                }
            }
        }.navigationBarTitle("News", displayMode: .inline)
    }
}
```

Этот листинг кода описывает экран списка новостей с заголовком и заложенными переходами на экран отдельной новости по тапу на элемент списка, как на рис. 10.2.

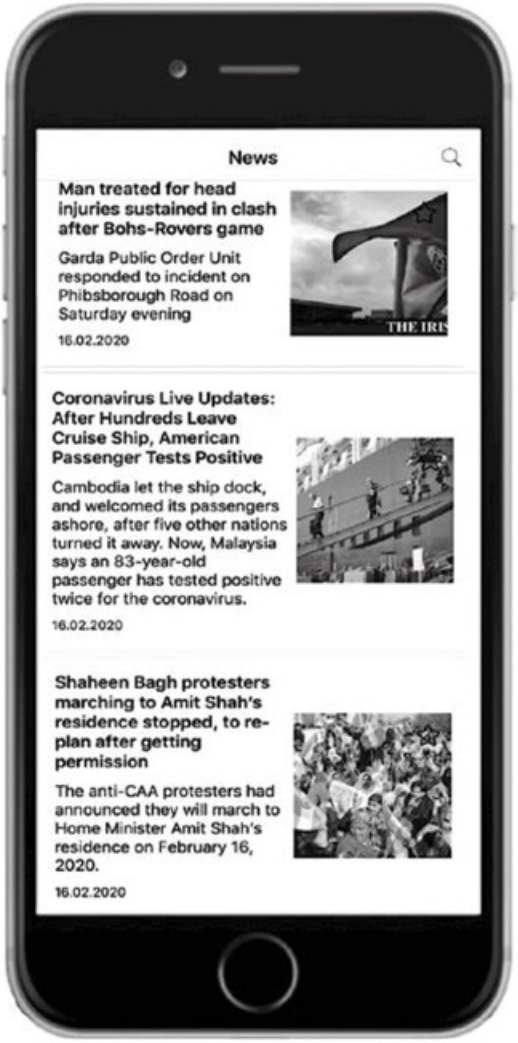


Рис. 10.2. Пример кода на SwiftUI

UI реализуется путем композиции и передачи декларативного блока выполнения для создания View внутрь родительского View с помощью механизма ViewBuilder. Сам UI строится внутри в зависимости от типа и настроек (встроенных или кастомизированных) родительского контейнера.

SwiftUI функционирует как декларативная обертка над теми же механизмами Autolayout и контролами UIKit iOS SDK. Фреймворк не дает прямого доступа к `UIView`, `UIViewController`, их свойствам, а также стеку навигации. Вся работа ведется через декларативные контролы-обертки (рис. 10.3).

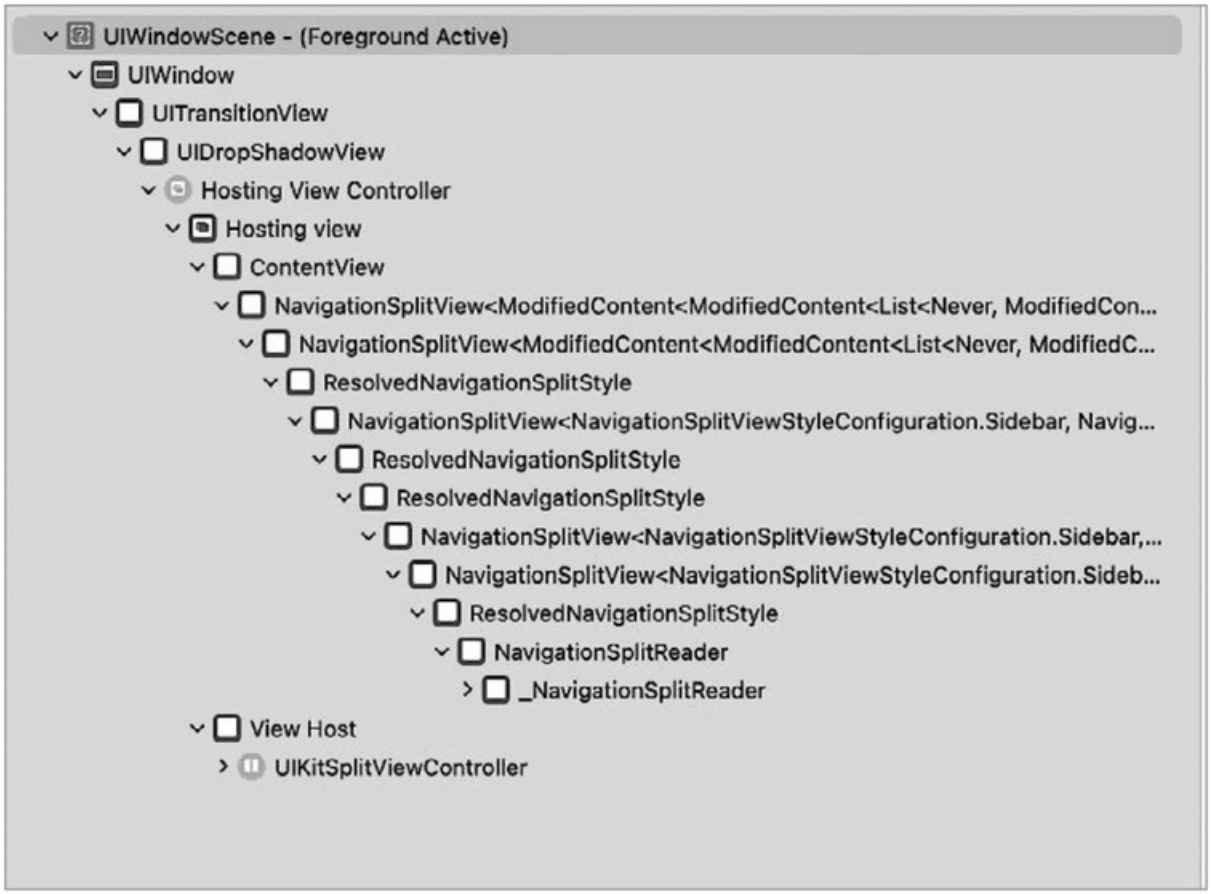


Рис. 10.3. Пример иерархии элементов в SwiftUI

Входной точкой для подключения SwiftUI к UI служат специальные Hosting View внутри корневого контроллера `UIHostingViewController`.

10.1.2. Нативный UI для Android

Теперь перейдем к Android. Работа с UI также построена на взаимодействии с контролами Android SDK.

Классический подход к верстке UI заключался в использовании файлов разметки на xml и их подключении к соответствующим View, экранам `Activity` или фрагментам Fragment (рис. 10.4).



Рис. 10.4. Пример UI на xml

Еще один из возможных подходов к реализации UI в Android состоит в использовании декларативного фреймворка JetpackCompose. В нем для верстки компонентов и создания экранов используются специальные декларативные функции Composable (рис. 10.5).



Рис. 10.5. Пример UI на Composable

На данный момент рекомендуется использовать декларативный фреймворк Jetpack Compose для верстки UI, стилизации, а также настройки навигации между компонентами.

Работа над Jetpack Compose началась в Google примерно в 2017 году, но впервые фреймворк был анонсирован на Google I/O в 2019 году, почти в то же время, что и SwiftUI. Но если SwiftUI не претерпел кардинальных изменений за время своего существования, то первая стабильная версия Compose появилась только в 2021 году.

Входной точкой в UI на Compose является метод `setContent` экрана `Activity` или группы View `ViewGroup` (листинг 10.5).

Листинг 10.5. Пример подключения Composable к Activity

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyApplicationTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    GreetingView(Greeting().greet())
                }
            }
        }
    }
}

@Composable
fun GreetingView(text: String) {
    Text(text = text)
}

@Preview
@Composable
fun DefaultPreview() {
    MyApplicationTheme {
        GreetingView("Hello, Android!")
    }
}
```

Это решение позволяет совмещать в одном коде Composable UI и UI, реализованный на xml или с помощью кода.

Jetpack Compose внутри работает с существующими контролами Android SDK, компонуя их в специальный `ComposeView` (листинг 10.6).

Листинг 10.6. Фрагмент преобразования Composable в ComposeView

```
internal fun ViewGroup.setContent(
        parent: CompositionContext,
        content: @Composable () -> Unit
): Composition {
        GlobalSnapshotManager.ensureStarted()
        val composeView =
                if (childCount > 0) {
                        getChildAt(0) as? AndroidComposeView
                } else {
                        removeAllViews(); null
                } ?: AndroidComposeView(context).also { addView(it.view,
                                        DefaultLayoutParams) }
        return doSetContent(composeView, parent, content)
}
```

DSL-конструктор Compose рекурсивно проходит по всем Composable-функциям, создает компоненты UI и добавляет их в родительские View. Для сохранения информации об иерархии и композиции, кэширования созданных компонентов, а также перестройки UI используется специальный объект Composer (листинг 10.7).

Листинг 10.7. Преобразование Composable

```
@Composable
fun Greeting(name: String) {
  Text("Hello, $name")
}

// При компиляции
fun Greeting(name: String, $composer: Composer) {
  Text("Hello, $name", $composer)
}
```

Composer рекурсивно передается по иерархии всем Composable.

10.2. Общий UI

Оба фреймворка, SwiftUI и Compose, заимствовали друг у друга (а также у Flutter и ReactNative) некоторые функциональные особенности и нюансы. В обоих используется механизм декларативного конструктора, который с помощью кода создает View из нативных контролов и компонует их между собой.

Но между ними есть и радикальные отличия:

- SwiftUI оперирует структурами, а Jetpack Compose — специальными функциями Composable;
- SwiftUI работает с механизмом ViewBuilder, а Jetpack Compose использует Composer и Kotlin DSL;

- нативные контролы iOS SDK и Android SDK разные.

Попробуем объединить работу с контролами данных SDK внутри общей логики и общего модуля по принципу `expect/actual` (рис. 10.6).

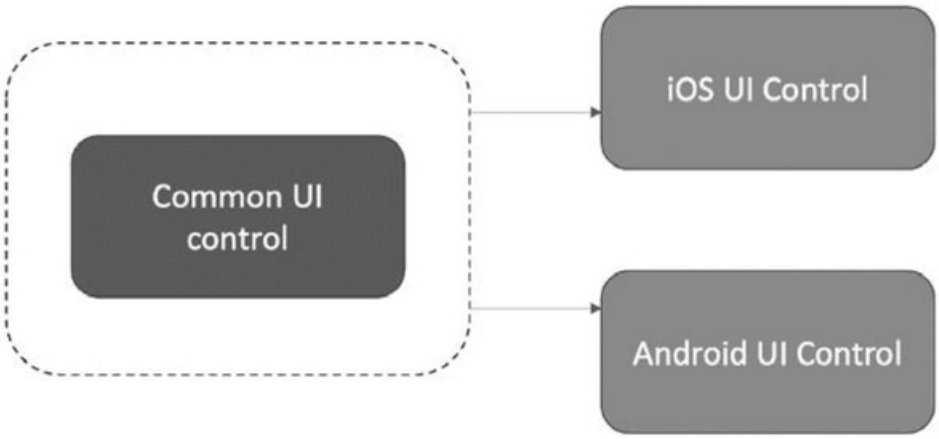


Рис. 10.6. Expect/actual-подход в создании общего контрола

Из этих общих контролов можно собрать общий экран, который будет работать под капотом и с нативным iOS, и с нативным Android, примерно как на рис. 10.7.

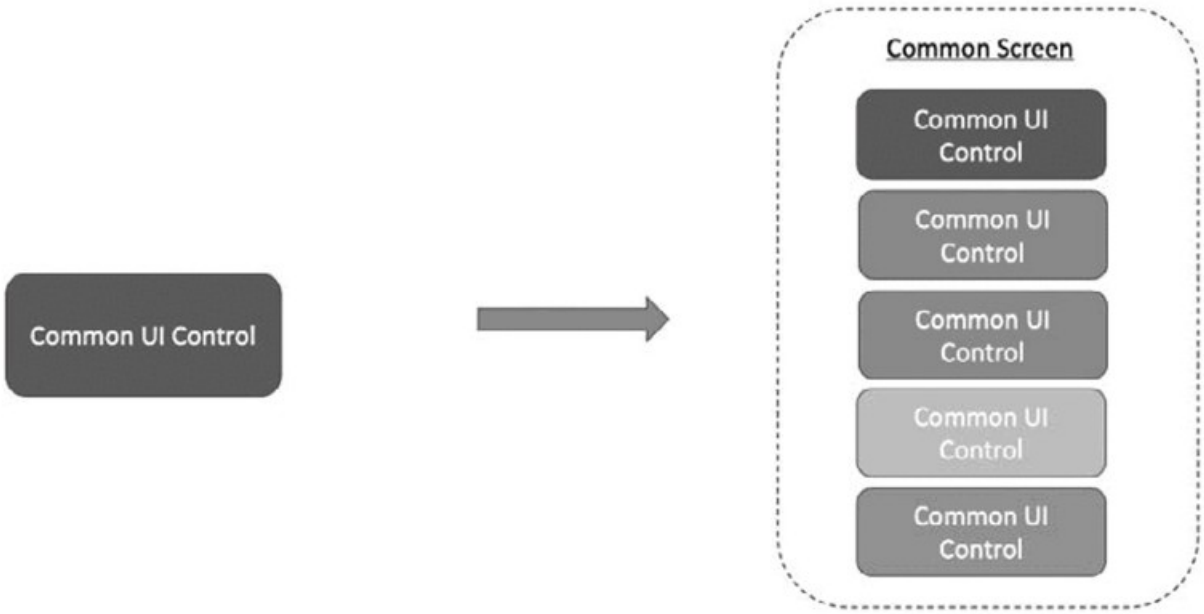


Рис. 10.7. Общий экран из общих контролов

Создание общих экранов из общих контролов можно реализовать несколькими способами:

- обернуть нативные контролы в общий (создать общую обертку в commonMain);

- обернуть общие контролы в нативные (подключить нативные контролы в androidMain/iOSMain);
- использовать общий интерфейс/протокол для View (на стороне нативного приложения).

10.2.1. Общая обертка для UI

Начнем с использования общего протокола/интерфейса и обертки wrapper. Рассмотрим схему ниже (рис. 10.8).

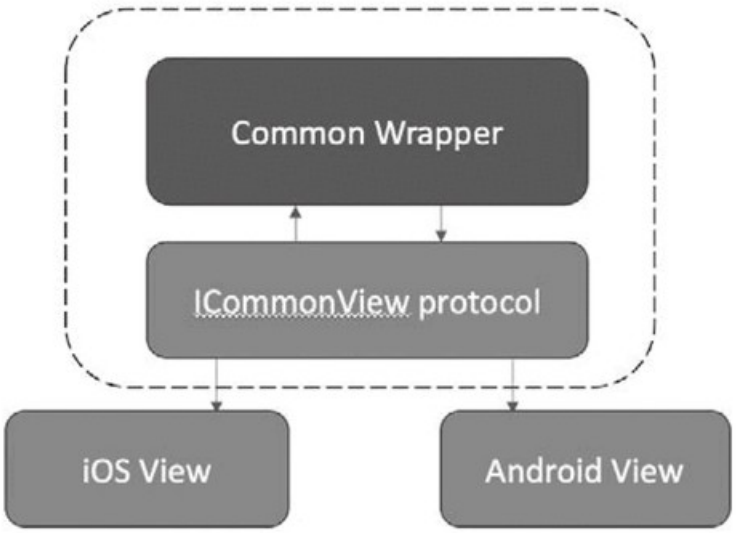


Рис. 10.8. Использование общей обертки для работы с нативными контролами

Common Wrapper представляет собой компонент, реализующий основную логику контрола и управляющий им через протокол `ICommonView`. Реализация протокола и подключение самих View идет на стороне iOS и Android соответственно.

Для декларативных фреймворков такой подход напрямую не подойдет, потребуется прокси, но для экранов на `Activity` и `UIViewController`, `ViewGroup` и `UIView` это вполне реализуемо (листинг 10.8).

Листинг 10.8. Пример использования общего интерфейса

```
interface CommonScreenView {
    fun setup()
    fun showNextScreen()
}

class CommoniOSVC : CommonScreenView {
    //...
    fun setup() {
        // iOS implementation
    }
}
```

```
class CommonAndroidActivity : CommonScreenView {
    //...
    fun setup() {
        // Android implementation
    }
}
```

В этом примере интерфейс `CommonScreenView` определяет общий протокол вызовов событий нативного View.

Обертка `CommonScreenWrapper` предоставляет шаблон управления компонентом `CommonScreenView` (листинг 10.9).

Листинг 10.9. Класс-обертка CommonScreenWrapper

```
interface CommonScreenView {
    fun setup()
    fun showNextScreen()
}

class CommonScreenWrapper {

    private var view: CommonScreenView? = null

    fun attach(view: CommonScreenView) { // <-- вызов из нативного приложения
        this.view = view
        create()
    }

    fun create() {
        view?.setup()
    }
//...
```

Ссылка на компонент View передается через метод `attach` со стороны нативного приложения.

При таком подходе мы можем настраивать элементы на стороне нативных приложений, а управлять ими из shared-модуля. С одной стороны, это дает нам относительную гибкость в реализации контролов на нативе, с другой — перекладывает на таргеты большую ответственность.

В случае использования базовых и стандартных контролов мы можем подключать в общую обертку компоненты не нативного приложения, а из iOSMain/androidMain таргетов shared-модуля.

10.2.2. Общий контрол в нативной обертке

Еще одним способом реализации общего UI является использование общего контрола, который подключается в нативную реализацию. В зависимости от сце-

нария разработки подключение общего контрола может происходить в iOSMain/androidMain или со стороны нативного приложения (рис. 10.9).

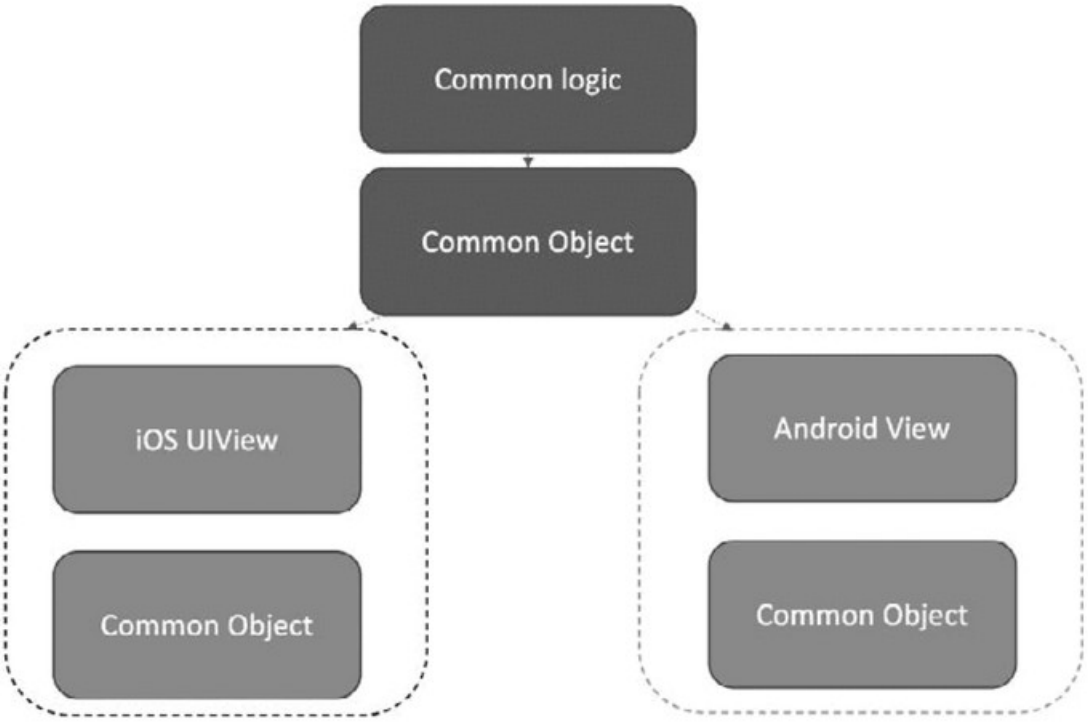


Рис. 10.9. Схема использования общего контрола в нативных приложениях

Такой подход можно назвать подключаемым UI (Pluggable UI).

В качестве примера реализации этого рассмотрим код из библиотеки Decompose (листинг 10.10).

Листинг 10.10. Пример подключаемого UI

```
import com.arkivanov.decompose.value.MutableValue
import com.arkivanov.decompose.value.Value
import com.arkivanov.decompose.value.update

class Counter {
  private val _state = MutableValue(State())
  val state: Value<State> = _state

  fun increment() {
    _state.update { it.copy(count = it.count + 1) }
  }

  data class State(val count: Int = 0)
}

// Android
import com.arkivanov.decompose.extensions.compose.jetbrains.subscribeAsState
```

```
@Composable
fun CounterContent(counter: Counter) {
  val state by counter.state.subscribeAsState()

  Column {
    Text(text = state.count.toString())

    Button(onClick = counter::increment) {
      Text("Increment")
    }
  }
}

// iOS
struct CounterView: View {
  private let counter: Counter

  @StateValue
  private var state: CounterState

  init(_ counter: Counter) {
//Подключение PropertyWrapper
    self.counter = counter
    _state = StateValue(counter.state)
  }

  var body: some View {
    VStack(spacing: 8) {
      Text(state.value.text)
      Button(action: counter.increment, label: { Text("Increment") })
    }
  }
}
```

Реализация объекта в commonMain включает в себя и общую логику, и модель состояния. Свойства реализованы с помощью хранителя состояний Value из библиотеки Decompose. Такой подход позволяет убрать зависимость от корутин Kotlin и от Kotlin Flow и сделать решение более гибким. Подробнее о библиотеке Decompose речь пойдет дальше в этой главе.

Концепция компонента перекладывает ответственность за реализацию отображения на нативные приложения. В определенной мере такой компонент больше соответствует компоненту бизнес-логики, чем части UI.

10.2.3. Подключение через alias

Если мы работаем с компонентами UI, которые известны нативным частям shared-модуля (androidMain/iOSMain), то можем воспользоваться подходом `expect/actual` для поддержки разных реализаций UI, но внутри shared-модуля (рис. 10.10).

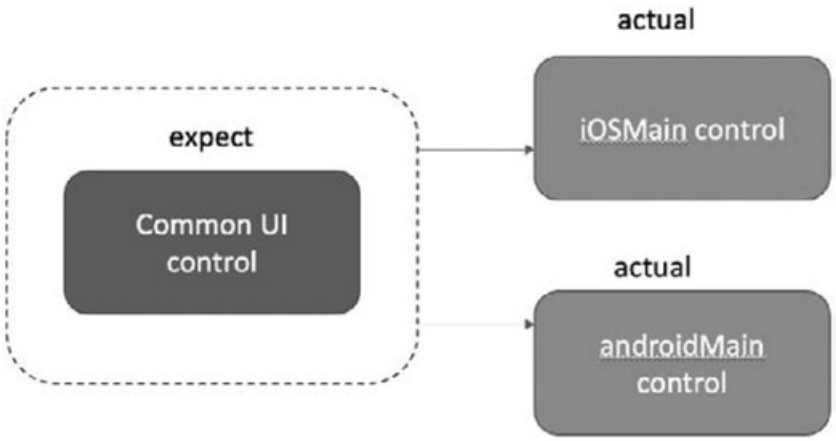


Рис. 10.10. Реализация общего UI с помощью expect/actual

Для этого в commonMain нужно объявить `expect`-декларацию UI-элемента, а в iOSMain/androidMain — реализацию `actual`. У классов для UI iOS SDK/Android SDK уже есть свои базовые классы, поэтому объединить контролы можно под общим типом через alias (листинг 10.11).

Листинг 10.11. Соединение контролов с помощью псевдонимов типов

```
//Common
expect class TextLabel

//Android
actual typealias TextLabel = TextView

//iOS
actual typealias TextLabel = UILabel
```

Стоит учитывать, что кроме присваивания типов потребуется также унифицировать логику разных типов, а это довольно сложно.

10.3. Compose Multiplatform

Все описанные подходы требуют серьезной проработки, подготовки к реализации и учета разнообразных нюансов механики работы UI на разных платформах. Поэтому при выборе решения стоит остановиться на наиболее эффективных вариантах. Одним из них является использование библиотеки Compose Multiplatform от компании JetBrains. Это готовый декларативный кросс-платформенный фреймворк для создания общего UI.

В настоящее время Compose Multiplatform (CMP) рекомендуется разработчиками технологии как дополнительный и опциональный слой приложения Kotlin Multiplatform (рис. 10.11).

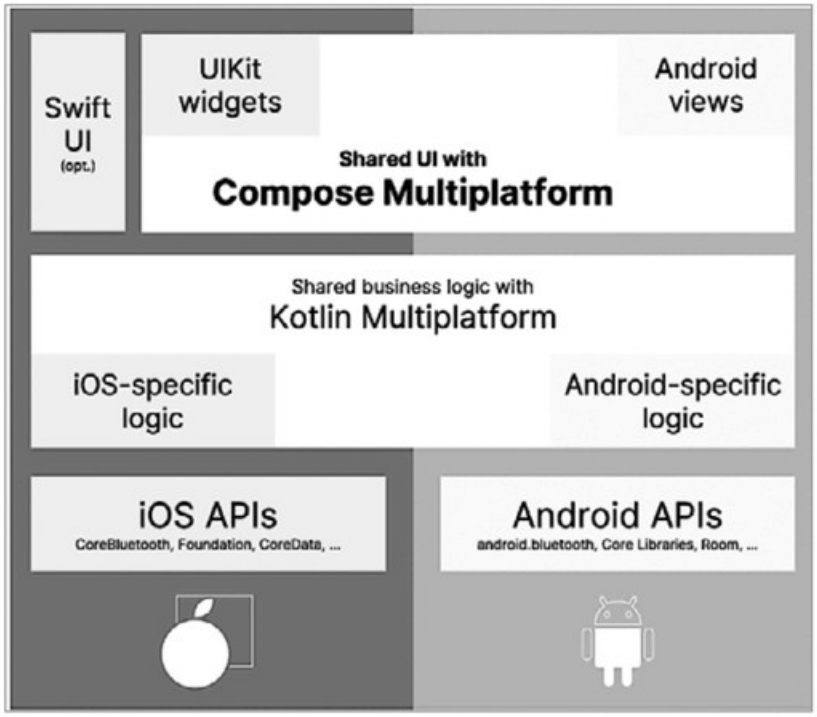


Рис. 10.11. Устройство приложения KMP

Как мы помним, изначально Kotlin Multiplatform был задуман как SDK кросс-платформенной разработки для создания модулей общего кода для разных таргетов. Он не предусматривал никаких средств для реализации слоя пользовательского интерфейса в отличие от Flutter и Xamarin Forms/MAUI. Тем не менее стремление к улучшению и развитию продукта подтолкнуло энтузиастов и контрибьюторов Kotlin Multiplatform к созданию собственных решений для разработки общего UI в кросс-платформенных приложениях KMP. Так появились такие технологии, как Decompose Аркадия Иванова, Moko Widgets (Ice Rock) и Redwood.

В 2020 году компания JetBrains приступила к разработке Compose Multiplatform для декстопа и веб. За основу был взят Jetpack Compose для Android. Android является нативным и родным таргетом для KMP-приложений, поэтому было проще и правильнее адаптировать и портировать именно существующие решения для Android, чем создавать нечто другое с нуля (как, например, в Flutter и Dart).

В конце 2021 года вышла версия Compose Multiplatform с поддержкой таргетов для desktop и Android. Версии для iOS и веб появились в 2022 году. В настоящее время этот продукт все еще находится в процессе разработки, несмотря на то что уже давно перешел в стадию beta.

Давайте посмотрим, как устроен и работает Compose Multiplatform. Для рендеринга пользовательского интерфейса Compose Multiplatform использует внутри Skia — мощную библиотеку от Google для работы с 2D-графикой. Универсальность Skia позволяет ей одинаково хорошо работать на разных платформах.

Стоит упомянуть, что Skia применяется как движок для графического рендеринга и в Flutter — другом известном кросс-платформенном UI-фреймворке от Google. Немаловажно, что Jetpack Compose тоже использует Skia для рендеринга, хотя и не напрямую.

Compose Multiplatform работает со Skiko — кросс-платформенной оберткой над Skia, специально адаптированной под поддержку разных версий языка Kotlin. Именно Skiko как прокси-объект позволяет использовать возможности Skia для рендеринга UI в кросс-платформенных приложениях.

Чтобы работать с iOS, Compose Multiplatform поддерживает интероп с нативным UIKit и SwiftUI для встраивания в Compose (рис. 10.12).



Рис. 10.12. Пример приложения на CMP со встроенной UI iOS

Этот подход мы рассмотрим для подключения нативных iOS-контролов в Compose View немного позже.

10.3.1. Подключение и настройка CMP в проекте

Начнем с подключения зависимостей в commonMain (листинг 10.12).

Листинг 10.12. Зависимости для Compose Multiplatform

```
[versions]
  compose = "1.5.4"
  compose-material3 = "1.1.2"
  androidx-activityCompose = "1.9.3"
  compose-multiplatform = "1.7.0"
  androidx-material = "1.12.0"
```

```
[libraries]
  androidx-activity-compose = { module = "androidx.activity:activity-compose",
                    version.ref = "androidx-activityCompose" }
  compose-ui = { module = "androidx.compose.ui:ui", version.ref = "compose" }
  compose-ui-tooling = { module = "androidx.compose.ui:ui-tooling",
                         version.ref = "compose" }
  compose-ui-tooling-preview = { module = "androidx.compose.ui:ui-tooling-
                                  preview", version.ref = "compose" }
  compose-material3 = { module = "androidx.compose.material3:material3",
                        version.ref = "compose-material3" }
  androidx-navigation-compose = { module = "org.jetbrains.androidx.
           navigation:navigation-compose", version.ref = "androidx-navigation" }

[plugins]
  compose-compiler = { id = "org.jetbrains.kotlin.plugin.compose",
                       version.ref = "kotlin" }
  composeMultiplatform = { id = "org.jetbrains.compose", version.ref =
                           "compose-multiplatform" }
```

Также нужно будет подключить не только зависимости в commonMain sourceSets, но и плагины (листинг 10.13).

Листинг 10.13. Подключение зависимостей в build.gradle

```
//Подключение в shared-модуль
sourceSets {
  commonMain.dependencies {
    implementation(compose.ui)
    implementation(compose.runtime)
    implementation(compose.foundation)
    implementation(compose.material3)
    implementation(compose.components.resources)
  }
}
// Подключение плагина в build.gradle.kts (shared-модуль)
plugins {
    //...
    alias(libs.plugins.composeMultiplatform)
    alias(libs.plugins.compose.compiler)
}


// Подключения плагина в build.gradle.kts (app)
plugins {
    //...
    alias(libs.plugins.compose.compiler).apply(false)
    alias(libs.plugins.composeMultiplatform) apply false
}
```

В качестве плагинов используется и специальный компилятор Compose.

10.3.2. Перенос и адаптация UI под Compose Multiplatform

Напишем адаптацию для экрана списка. Если у нас есть код для Android-приложения на Compose, то достаточно скопировать его в commonMain и обновить импорты. В нашем учебном приложении мы не используем сложный UI и/или анимации, поэтому дополнительных действий с нашей стороны не потребуется. Мы поддерживаем iOS, поэтому разберем код наших элементов подробнее.

Элемент списка представляет собой горизонтальный контейнер `Row` (аналог `HStack` SwiftUI). Внутри располагается элемент изображения `Image` и вертикальный контейнер `Column` (аналог `VStack`) с тремя текстовыми полями `Text` (листинг 10.14).

Листинг 10.14. Composable для элемента списка

```
@Composable
fun NewsListItemView(item: NewsItem,
                     modifier: Modifier = Modifier,
                     onFavoriteClick: (NewsItem) -> Unit = {}) {
// 1 Горизонтальный контейнер
    Row(
        modifier = modifier
            .fillMaxWidth()
            .padding(8.dp, 8.dp)
            .background(Color.White),
        verticalAlignment = Alignment.Top,
        horizontalArrangement = Arrangement.SpaceAround,
    ) {
// 2 Контейнер для наложения элементов друг на друга
        Box(Modifier.width(120.dp).height(120.dp).padding(0.dp, 0.dp, 8.dp,
                                0.dp)) {
            Image(
                painter = rememberImagePainter(item.urlToImage.orEmpty()),
                contentDescription = "image",
                modifier = Modifier.fillMaxSize(),
                contentScale = ContentScale.Crop
            )
// 3 Кнопка с поддержкой добавления в избранное
            IconButton(onClick = { onFavoriteClick.invoke(item) }) {
                Icon(
                    imageVector = if (item.isFavorite) Icons.Filled.
                                  Favorite else Icons.Filled.FavoriteBorder,
                    contentDescription = "favorite",
                    modifier = Modifier.width(50.dp).height(50.dp),
                )
            }
        }
// 4 Вертикальный контейнер с текстовыми полями
        Column {
```

```
            Text(text = item.title.orEmpty(), style = MaterialTheme.typography.
                        titleMedium)
            Text(text = item.content.orEmpty(), style = MaterialTheme.
                        typography.titleSmall, maxLines = 3)
            Text(text = item.publishedAt.orEmpty(), style = MaterialTheme.
                        typography.bodyMedium)
        }
    }
}
```

В качестве параметров передадим в функцию элемент `NewsItem`, а также обработчик события добавления в избранное `onFavoriteClick`. Вызов события обработчика можно прикрепить к кнопке IconButton. Для того чтобы кнопка располагалась поверх изображения, объединим их в контейнер Box.

Обратите внимание, что мы используем функцию `rememberImagePainter` для поддержки загрузки изображения из url. В Android Jetpack Compose для этих целей применяется нативный функционал. Для Compose Multiplatform потребуется либо самописный Composable, либо сторонняя библиотека (листинг 10.15).

Листинг 10.15. Подключение сторонней библиотеки для загрузки изображений асинхронно

```
sourceSets {
  commonMain.dependencies {
    implementation(compose.components.resources)
    api("io.github.qdsfdhvh:image-loader:1.7.1")
  }
}

import com.seiko.imageloader.rememberImagePainter

// Подключение Composable
Image(
    painter = rememberImagePainter(item.urlToImage.orEmpty()),
    contentDescription = "image",
    modifier = Modifier.fillMaxSize(),
    contentScale = ContentScale.Crop
)
```

Добавим `Composable` для экрана списка. В качестве параметров передадим в `NewsListView` список элементов `List<NewsItem>`, а также два события: `onClick` для выбора элемента и `onFavoriteClick` для добавления в избранное элемента (листинг 10.16).

Листинг 10.16. Composable для экрана новостей

```
@Composable
fun NewsListView(items: List<NewsItem>,
                 onClick: (NewsItem) -> Unit,
                 onFavoriteClick: (NewsItem) -> Unit) {
// 1 Контейнер для списка
```

```
    LazyColumn(
        contentPadding = // 1.
            PaddingValues(horizontal = 16.dp, vertical = 8.dp),
        modifier = Modifier.background(Color.White)
    ) {
// 2 Передача items в функцию-расширения
        items(items) {
// 3 Вызов создания элемента списка
            NewsListItemView(it,
// 4 Блок добавления в избранное
                onFavoriteClick = onFavoriteClick,
// 5 Подключение обработчика через Modifier.clickable
                modifier = Modifier.clickable {
                    onClick.invoke(it)
                })
        }
    }
}
```

Для списка используем контейнер `LazyColumn`. Рендеринг, подключение и создание элементов реализуются с помощью функции-расширения items, в которую передаем список `List<NewsItem>`.

Для того чтобы сделать этот элемент полностью кликабельным, нам нужно подключить расширение `Modifier.clickable` и передать туда вызов события. Мы используем `onClick: (NewsItem) -> Unit`, и в качестве параметра в блок передается тот элемент `NewsItem`, который мы выбираем.

Теперь соединим `Composable` экрана с бизнес-логикой. Для сохранения состояния мы передадим `ViewModel` из внешнего компонента, в нашем случае — экрана `Activity`. Создадим еще один `Composable`, в котором подключим подписку на свойство с данными (листинг 10.17).

Листинг 10.17. Подключение бизнес-логики к экрану

```
@Composable
fun NewsListScreen(viewModel: NewsViewModel,
        onClick: (NewsItem)->Unit) {

// 1 Загрузка данных в onLaunch
    LaunchedEffect(Unit) {
        println("onStart")
        viewModel.loadNews()
    }
// 2 Подписка на данные
    val newsList: List<NewsItem>? by viewModel.newsFlow.
collectAsState(emptyList())
// 3 Передача данных и событий в Composable
    NewsListView(newsList.orEmpty(), onClick, {
        viewModel.onFavoriteClick(it)})
}
```

Запрос данных из `ViewModel` вызываем в событии `onLaunch` и блоке `LaunchedEffect`.

10.4. Навигация в проектах с Compose Multiplatform

Навигация, конфигурация экранов, управление состоянием — это одни из ключевых моментов при реализации приложений, особенно кросс-платформенных. Если мы реализуем UI нативно, то логично реализовать нативно и навигацию между экранами с передачей данных. Для Android-приложения с UI на xml вызов новых экранов реализуется через специальные намерения Intent.

Для Android приложений с UI на Jetpack Compose используется либо тот же переход через Intent между `Activity`, либо специальная библиотека `Navigation` (листинг 10.18).

Листинг 10.18. Пример вызова Activity

```
val intent = Intent(context, NewsItemActivity::class.java).apply {
    putExtra("news_item", newsItem)
}
context.startActivity(intent)
```

На iOS мы используем либо добавление экрана в стек навигации `UINavigationViewController` (листинг 10.19),

Листинг 10.19. Переход на UIViewController

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
  tableView.deselectRow(at: indexPath, animated: true)

  let selectedItem = newsItems[indexPath.row]

  // Создание и переход к NewsItemVC
  let newsItemVC = NewsItemVC(newsItem: selectedItem)

  // Если есть UINavigationController:
  navigationController?.pushViewController(newsItemVC, animated: true)

  // Если нет UINavigationController:
  // present(newsItemVC, animated: true)
}
```

либо `NavigationLink` и `NavigationStack` для SwiftUI (листинг 10.20).

Листинг 10.20. Фрагмент навигации в View SwiftUI

```
NavigationStack {
            List(newsItems) { item in
                // Навигация с помощью NavigationLink
                NavigationLink(value: item) {
                    NewsRow(item: item)
```

```
            }
        }
        .navigationTitle("Новости")
        .navigationDestination(for: NewsItem.self) { item in
            NewsItemRow(item: item)
        }
    }
```

Все эти решения стандартны и за исключением библиотеки Navigation не требуют установки сторонних зависимостей.

При использовании Compose Multiplatform встает вопрос, какую же реализацию все-таки стоит использовать и как это сделать правильно. Мы рассмотрим три решения: PreCompose, библиотеку навигации от JetBrains и Decompose (подраздел 10.7.2).

10.4.1. Навигация на PreCompose

Библиотека PreCompose (https://github.com/Tlaster/PreCompose) реализована по принципам навигации Jetpack Navigation, с поддержкой `ViewModel` и жизненного цикла объектов. Авторы постарались максимально полно адаптировать нативную навигацию Android под кросс-платформенные приложения, в результате миграция с нативного решения получается быстрой и безболезненной.

Начнем с установки библиотеки (листинг 10.21).

Листинг 10.21. Подключение PreCompose

```
sourceSets {
        commonMain.dependencies {
            api("moe.tlaster:precompose:*")
        }
    }
```

Добавим зависимость в commonMain как API.

Нам нужна единая входная точка, которую мы будем вызывать из `Activity` на стороне Android и `UIViewController` на стороне iOS. Создадим специальный Composable, внутри которого декларативно разместим конструктор навигации `PreComposeApp` (листинг 10.22).

Листинг 10.22. Входная точка PreCompose

```
@Composable
fun App() {

  PreComposeApp {
    //...
  }
}
```

В блоке `PreComposeApp` настроим навигатор, с помощью которого будем осуществлять переходы. Также нам потребуется получить доступ к экранам из стека навигации `backStackEntries`. На основе информации о состоянии стека мы можем получать доступ к текущему экрану (листинг 10.23).

Листинг 10.23. Настройка навигации

```
PreComposeApp {
    // Навигатор
    val navigator = rememberNavigator()

    // VM стартового экрана
    val vm: NewsViewModel = KoinDIFactory.resolve(NewsViewModel::class)!!

    // Стек экранов
    val backStackEntry by navigator.currentEntry.collectAsState(null)

    // Текущий стартовый экран
    val currentScreen = NewsScreen.valueOf(
        backStackEntry?.route?.route ?: NewsScreen.NewsList.name
    )
    // Определяем доступность кнопки "назад"
    val canNavigateBack by navigator.canGoBack.collectAsState(false)
    Scaffold(
        topBar = {
            AppBar(
                currentScreen = currentScreen,
                canNavigateBack = canNavigateBack,
                navigateUp = { navigator.goBack() }
            )
        }
    ) {//NavHost...}
}
```

`backStackEntry` — реактивное свойство, с помощью которого мы можем отслеживать изменения в стеке экранов.

С помощью `rememberNavigator` подключаем хранение состояния навигатора и подписку на изменения (листинг 10.24).

Листинг 10.24. Функция rememberNavigator

```
/**
 * Creates a [Navigator] that controls the [NavHost].
 *
 * @see NavHost
 */
@Composable
fun rememberNavigator(): Navigator {
  val stateHolder = LocalStateHolder.current
  return stateHolder.getOrPut("Navigator") {
    Navigator()
  }
}
```

ВАЖНО! `ViewModel` инициализируется на уровне над хранителем состояний приложения и только затем передается в Composable экрана.

Создадим Composable Scaffold для настройки структуры этого экрана. В качестве заголовка используем самописный Composable AppBar, куда передадим название экрана, блок вызова перехода по навигатору назад и флаг показа кнопки возврата (если нет экранов в стеке, кроме текущего, то кнопку не показываем). С кодом AppBar можно ознакомиться в листинге 10.25.

Листинг 10.25. Настройка AppBar

```
@Composable
fun AppBar(
    currentScreen: NewsScreen,
    canNavigateBack: Boolean,
    navigateUp: () -> Unit,
    modifier: Modifier = Modifier
) {
    TopAppBar(
        title = { currentScreen.name },
        modifier = modifier,
        navigationIcon = {
            if (canNavigateBack) {
                IconButton(onClick = navigateUp) {
                    Icon(
                        imageVector = Icons.Filled.ArrowBack,
                        contentDescription = "back"
                    )
                }
            }
        })
}
```

Займемся переходами. У нас есть два экрана: список новостей и экран деталей новости. При нажатии на элемент списка мы передаем информацию с первого экрана на второй.

В блоке нашего Scaffold создадим `NavHost` (листинг 10.26).

Листинг 10.26. Настройка NavHost

```
NavHost(
    navigator = navigator,
    navTransition = NavTransition(),
    // The start destination
    initialRoute = NewsScreen.NewsList.name
) {
    scene(
        route = NewsScreen.NewsList.name,
        navTransition = NavTransition()
    ) {
        NewsListScreen(vm) { item ->
```

```
                navigator.navigate(NewsScreen.NewsItem.name)
            }
        }
        scene(route = NewsScreen.NewsItem.name) {
                NewsItemDataView(NewsItemVM(item))    }
}
```

В NavHost передадим объект навигатора, новый экземпляр механики переходов NavTransition, а также информацию о стартовом экране.

Переход на каждый из экранов опишем с помощью блока scene, в который передадим ключ экрана. Для этого можно создать `enum` со строковыми значениями (листинг 10.27).

Листинг 10.27. Пример enum с ключами для экранов

```
enum class NewsScreen {
  NewsList,
  NewsItem
}
```

Для передачи данных между экранами в Android используются механизмы сериализации и парселизации, а также специальные Bundle — хранилища состояний и данных. В iOS данные присваиваются определенным полям следующего экрана. В качестве общего решения для передачи данных между экранами в commonMain реализуем специальный объект-хранилище, где информация находится в сериализованном виде (листинг 10.28).

Листинг 10.28. Объект для хранения и передачи данных в сериализованном виде

```
object Coder {
    val map = hashMapOf<String,String>()

    inline fun<reified T> save(key: String, data: T) {
        map[key] = encode(data)
    }

    inline fun<reified T> get(key: String):T? {
        return  decode(map[key].orEmpty())
    }

    inline fun<reified T> encode(data: T):String {
        return Json.encodeToString(data)
    }

    inline fun<reified T> decode(json: String):T? {
        return try {
            Json.decodeFromString(json)
        }catch (e: Exception) {
            null
        }
    }
}
```

Для кодирования и декодирования данных в JSON в этом решении используется механизм сериализации `kotlinx.serialization`.

Теперь подключим кодировщик в навигационный хост (листинг 10.29).

Листинг 10.29. Подключение сохранения и получения данных в навигацию

```
NavHost(
    navigator = navigator,
    navTransition = NavTransition(),
    // The start destination
    initialRoute = NewsScreen.NewsList.name
) {
    scene(
        route = NewsScreen.NewsList.name,
        navTransition = NavTransition()
    ) {
        NewsListScreen(vm) { item ->
            Coder.save("data", item)

            navigator.navigate(NewsScreen.NewsItem.name)
        }
    }
    scene(route = NewsScreen.NewsItem.name) {
        Coder.get<NewsItem>("data")?.let { item ->
            NewsItemDataView(NewsItemVM(item))
        }
    }
}
```

При запросе навигации кликом по строке новости мы сохраняем выбранный элемент в кодировщике, а при запросе с экрана единичной новости достаем данные из хранилища и передаем их в создаваемую `ViewModel` нового экрана.

Мы получили довольно легковесное решение, которое удобно использовать в приложении KMP. Конфигурацию экранного модуля стоит соединить с предполагаемой фабрикой `ViewModel`, которая будет отдавать нужный экземпляр в зависимости от типа экрана (листинг 10.30).

Листинг 10.30. Фабрика ViewModel

```
class ViewModelCreator {

  fun createViewModel(type: NewsScreen): BaseViewModel {
    return when (type) {
      NewsScreen.NewsList -> KoinDIFactory.resolve(NewsViewModel::class)!!
      NewsScreen.NewsItem -> NewsItemVM(Coder.get<NewsItem>("data")!!)
    }
  }
}
```

Теперь займемся подключением входного компонента App к нативным приложениям. На Android добавим вызов App в метод `setContent` (листинг 10.31).

Листинг 10.31. Подключение к Android

```
// Подключение к Android App
class MainActivity : ComponentActivity(){

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    KoinDIFactory.setupKoin {
      androidContext(applicationContext)
    }
    setContent {
      App()
    }
  }
}
```

Для iOS механизм несколько сложнее. Используем специальную функцию `ComposeUIViewController` для создания `UIViewController` (листинг 10.32).

Листинг 10.32. ComposeUIViewController и ее использование

```
fun ComposeUIViewController(content: @Composable () -> Unit): UIViewController =
    ComposeUIViewController(configure = {}, content = content)

fun ComposeUIViewController(
    configure: ComposeUIViewControllerConfiguration.() -> Unit = {},
    content: @Composable () -> Unit
): UIViewController = ComposeHostingViewController(
    configuration = ComposeUIViewControllerConfiguration().apply(configure),
    content = content,
)


// Подключение к iOS
fun MainViewController() = ComposeUIViewController { App() }
```

В качестве параметра передаем Composable App.

Теперь нужно подключить `MainViewController` на стороне iOS-приложения. Для того чтобы вызвать `UIViewController` в SwiftUI, используем специальный протокол `UIViewControllerRepresentable` (листинг 10.33).

Листинг 10.33. Подключение UIViewController через UIViewControllerRepresentable

```
struct ComposeView: UIViewControllerRepresentable {

    func makeUIViewController(context: Context) -> UIViewController {
        KoinDIFactory.shared.setupKoin { _ in

        }
        return MainViewControllerKt.MainViewController()
```

```
    }

    func updateUIViewController(_ uiViewController: UIViewController,
                                context: Context) {}
}
```

Реализуем протокол в структуре `ComposeView`. В методе `makeUIViewController` нужно вызвать функцию `MainViewController()`. Так как это глобальная функция Kotlin, расположенная в файле `MainViewController.kt`, обращаться к ней нужно через нотацию: `MainViewControllerKt.MainViewController`.

Подключим `ComposeView` в нативный App SwiftUI как корневой View (листинг 10.34).

Листинг 10.34. Подключение ComposeView

```
@main
struct iOSApp: App {
    var body: some Scene {
       WindowGroup {
            ComposeView()
       }
    }
}
```

10.4.2. Нативная навигация от JetBrains

Эта библиотека не входит по умолчанию в Compose Multiplatform, и ее необходимо устанавливать отдельно (листинг 10.35).

Листинг 10.35. Установка библиотеки навигации

```
[versions]
  androidx-navigation = "2.7.0-alpha06"

[libraries]
  androidx-navigation-compose = { module = "org.jetbrains.androidx.navigation:
           navigation-compose", version.ref = "androidx-navigation" }
```

В настоящее время библиотека находится в разработке на стадии alpha, поэтому использовать ее в реальных приложениях не рекомендуется. Но это никак не мешает проверить ее возможности в тестовых или пет-проектах.

Алгоритм настройки навигации достаточно прост:

1. Нужно создать `enum` для ключей к экранам, которые будут использоваться в графе навигации.
2. Добавить `NavHostController` и настроить его параметры.
3. Добавить граф навигации с описанием переходов внутри блока `NavHost`.

Enum будет похож на тот, который мы использовали для PreCompose. Нам только понадобится добавить для него конструктор с передачей параметра — id ресурса для заголовка и ключа (листинг 10.36).

Листинг 10.36. Enum для описания экранов

```
enum class NewsScreen (val title: StringResource)
 {
    NewsList(Res.string.news_list),
    NewsItem(Res.string.news_item)
}
```

Библиотека от JetBrains и PreCompose используют подход, как в навигации Android, поэтому код настройки будет практически идентичным (листинг 10.37).

Листинг 10.37. Настройка навигации

```
NavHost(
    navController = navController,
    startDestination = NewsList.Start.name,
    modifier = Modifier
        .fillMaxSize()
        .verticalScroll(rememberScrollState())
        .padding(innerPadding)
) {
  composable(route = NewsScreen.NewsList.Start.name) {
    NewsListScreen(vm) { item ->
      Coder.save("data", item)

      navigator.navigate(NewsScreen.NewsItem.name)
    }
  }
  composable(route = route = NewsScreen.NewsItem.name) {
    Coder.get<NewsItem>("data")?.let { item ->
      NewsItemDataView(NewsItemVM(item))
    }
  }
}
```

Обратите внимание на отличие в названии DSL-функции: не screen, а composable.

Навигация от JetBrains сейчас по умолчанию не поддерживает:

- работу с диплинками;
- интеллектуальный жест назад;
- собственный инструментарий для передачи данных между экранами.

Для решения последней задачи мы можем использовать тот же подход с объектом Coder, что и для PreCompose.

Применим на практике. Добавим еще один экран для просмотра деталей новости и соединим экраны с помощью механизма навигации на PreCompose:

1. Добавим `composable` для вывода данных по элементу (листинг 10.38).

Листинг 10.38. Composable для экрана деталей новости

```
@Composable
fun NewsItemView(
        modifier: Modifier = Modifier,
        image: Painter,
        title: String = "",
        text: String = ""
) {
        Column(modifier = modifier.padding(horizontal = 20.dp,
                vertical = 20.dp)) {
                Image(painter = image,
                        contentDescription = "image",
                        modifier = Modifier.height(200.dp))
                Text(title, style = TextStyle.Default)
                Text(text)
        }
}
```

2. В вертикальном контейнере Column расположим изображение и текстовые поля. Для изображения ограничим высоту в 200dp.

   Нам потребуется `ViewModel` для хранения состояния экрана (листинг 10.39).

Листинг 10.39. ViewModel для экрана новости

```
class NewsItemVM(item: NewsItem?) : ViewModel() {
  private val _item: MutableSharedFlow<NewsItem> = MutableSharedFlow(1,
onBufferOverflow = BufferOverflow.DROP_OLDEST)
  val model = _item.asSharedFlow()
}
```

   Регистрируем `ViewModel` в DI-компоненте.

3. Добавим Composable для подключения `ViewModel` к UI (листинг 10.40).

Листинг 10.40. Подключение ViewModel к Composable

```
@Composable
fun NewsItemDataView(viewModel: NewsItemVM) {
    val data by viewModel.model.collectAsState(null)

// Подключение элемента
    NewsItemView(
        image = rememberImagePainter(data?.urlToImage.orEmpty()),
        title = data?.title.orEmpty(),
        text = data?.content.orEmpty()
    )
}
```

4. Соединим экраны с помощью навигации. Создадим и настроим граф навигации, как в разделе 10.4.1.

5. Подключим реализованные Composable для экранов в граф навигации (листинг 10.41).

Листинг 10.41. Подключение экранов

```
NavHost(
    navigator = navigator,
    navTransition = NavTransition(),
    // The start destination
    initialRoute = NewsScreen.NewsList.name
) {
    scene(
        route = NewsScreen.NewsList.name,
        navTransition = NavTransition()
    ) {
        NewsListScreen(vm) { item ->
            Coder.save("data", item)
            navigator.navigate(NewsScreen.NewsItem.name)
        }
    }
    scene(route = NewsScreen.NewsItem.name) {
        Coder.get<NewsItem>("data")?.let { item ->
            NewsItemDataView(NewsItemVM(item))
        }
    }
}
```

6. Инициализируем `ViewModel` для экрана деталей новости при создании экрана. Данные получаем из объекта Coder. Мы также можем брать `ViewModel` из DI и передавать туда информацию из Coder.
7. Добавим входные точки для Android и iOS (листинг 10.42).

Листинг 10.42. Входные точки

```
// Подключение к Android App
class MainActivity : ComponentActivity(){

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    KoinDIFactory.setupKoin {
      androidContext(applicationContext)
    }
    setContent {
      App()
    }
  }
}

// Подключение к iOS
fun MainViewController() = ComposeUIViewController { App() }

struct ComposeView: UIViewControllerRepresentable {
```

```
    func makeUIViewController(context: Context) -> UIViewController {
        KoinDIFactory.shared.setupKoin { _ in

        }
        return MainViewControllerKt.MainViewController()
    }

    func updateUIViewController(_ uiViewController: UIViewController,
context: Context) {}
}
```

8. Проверим результат (рис. 10.13).

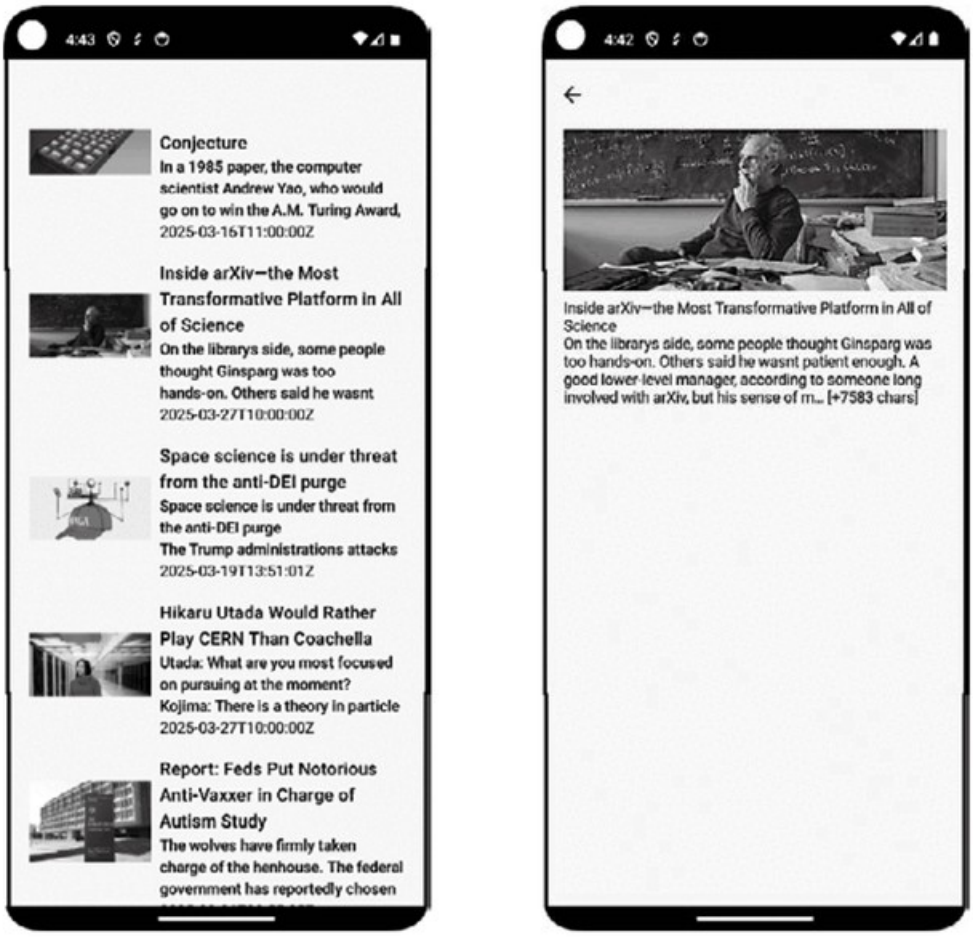


Рис. 10.13. Работа навигации

10.5. Подключение нативного кода к Compose Multiplatform

Код `Compose` поддерживает прямой интероп с `UIViewController` или `UIView`, то есть с UI UIKit. Для того чтобы подключить реализованные таким образом компоненты UI, нужно создать структуры `UIViewControllerRepresentable` или `UIViewRepresentable` (листинг 10.43).

Листинг 10.43. Подключение нативных компонентов к iOS

```
struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIViewController {
        AppViewController() // Kotlin ViewController
    }

    func updateUIViewController(_ uiViewController: UIViewController,
                                context: Context) {}
}


@main
struct iOSApp: App {
         var body: some Scene {
                 WindowGroup {
                       ComposeView()
        }
         }
}
```

Таким образом мы подключаем Compose к SwiftUI.

В некоторых задачах требуется обратное преобразование и подключение SwiftUI в код Compose. Представим, что мы хотим унифицировать работу с картами и использовать свой контрол в общем коде. Создадим View SwiftUI и подключим туда контрол карты `MapView` (листинг 10.44).

Листинг 10.44. View SwiftUI для работы с картой

```
import MapKit
import SwiftUI

struct MyMapView: View {
  @State private var region = MKCoordinateRegion(
    center: CLLocationCoordinate2D(latitude: 51.507222, longitude: -0.1275),
    span: MKCoordinateSpan(latitudeDelta: 0.5, longitudeDelta: 0.5)
  )

  var body: some View {
// Карта SwiftUI
    Map(coordinateRegion: $region)
      .frame(width: 300, height: 300)
  }
}
```

Теперь нам нужно преобразовать View SwiftUI для использования с UIKit. Для этого обернем контрол в `UIHostingController` — специальный `UIViewController`, поддерживающий внедрение SwiftUI в `HostingView` (листинг 10.45).

Листинг 10.45. Подключение SwiftUI к UIViewController

```
func myMapViewFactory() -> UIViewController {
    let myMapView = MyMapView() // Создаем SwiftUI View
    return UIHostingController(rootView: myMapView) // Внедряем
                                                    в UIHostingController
}
```

На стороне commonMain создадим функцию-фабрику. В качестве параметра будем передавать блок создания `UIViewController` карты (листинг 10.46).

Листинг 10.46. Функция-фабрика для карты

```
fun MapViewController(
    // Здесь будет вызов создания UIViewController с SwiftUI View
    mapFactory: () -> UIViewController
) = ComposeUIViewController {
    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
    ) {
        Text("Hello from compose")
        Text("Rendering map from SwiftUI below")

        // Rendering the SwiftUI View
        UIKitViewController(
            modifier = Modifier.size(300.dp).border(1.dp, Color.Black),
            factory = mapFactory,
            update = {}
        )

        Text("This is compose again")
    }
}
```

`UIKitViewController` — это composable-функция для подключения `UIViewController` и встраивания в Compose.

Создадим в iOS-приложении структуру `UIViewControllerRepresentable` (листинг 10.47).

Листинг 10.47. Обертка для фабрики карты

```
struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIViewController {
        MapViewController(mapFactory: myMapViewFactory)
    }

    func updateUIViewController(_ uiViewController: UIViewController,
                                context: Context) {}
}
```

Вызовем composable-функцию `MapViewController` из commonMain в методе `makeUIViewController ComposeView`, в качестве параметра передадим вызов функции `myMapViewFactory` для создания `UIHostingViewController`.

Мы сможем использовать `MapViewController` в commonMain, соединив с другой общей логикой для карты.

При необходимости через функцию-фабрику в commonMain можно передавать `ViewModel` в качестве дополнительного параметра (листинг 10.48).

Листинг 10.48. Подключение ViewModel через фабрику

```
fun MapViewController(
    mapFactory: (viewModel: MapViewModel) -> UIViewController
) = ComposeUIViewController {
    val viewModel: MapViewModel = // создание ViewModel
    UIKitViewController(
        modifier = Modifier,
        factory = { mapFactory(viewModel) }, // передаем в функцию-фабрику
        update = {}
    )
}
```

10.6. Server Driven UI

Особенность декларативных фреймворков для UI состоит в том, что иерархия визуальных элементов, их свойства, события и настройки задаются с помощью кода. Это означает, что они могут быть созданы автоматически с помощью собственных DSL-конструкторов, и это позволяет использовать их вместе с подходами Server-Driven UI (SDUI)/Backend-Driven UI (BDUI)/Data-Driven UI.

Все эти подходы подразумевают, что визуальная составляющая экрана в виде контракта данных либо задается на удаленном сервере, либо создается локально из набора data-моделей. Полученные модели данных преобразуются с помощью специального инструментария в UI (рис. 10.14).

Обычно данные передаются в формате JSON от сервера, затем декодируются и компонуются на стороне клиента.

Чтобы реализовать SDUI на стороне клиента, нужно ответить на следующие вопросы и решить связанные с ними задачи:

- Как обработать и во что преобразовать исходный JSON?
- Как оптимизировать создание UI?

В случае кросс-платформы возникает еще один вопрос: как сделать так, чтобы UI на нативных таргетах выглядел практически одинаково?

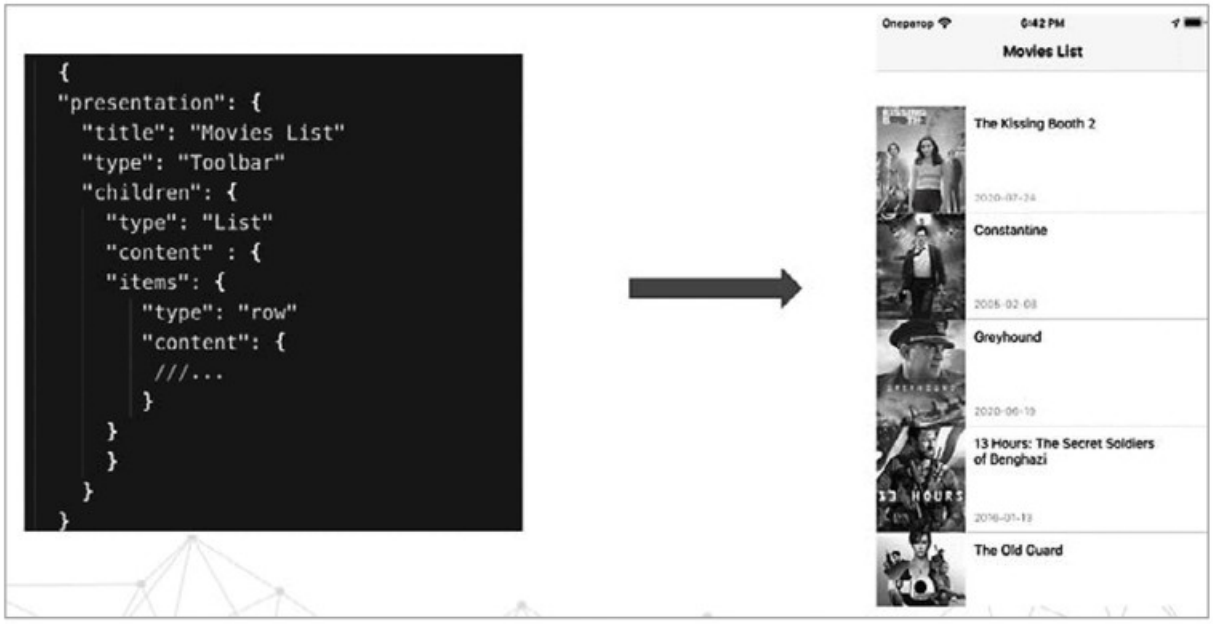


Рис. 10.14. Преобразование JSON в UI экрана списка данных

Начнем с решения первых двух задач с помощью создания специального обработчика и конвертера для компонентов SDUI.

На первый взгляд этот компонент похож на обычный декодер JSON. Но на деле JSON нужно не только декодировать, но и преобразовать в компоненты с определенными свойствами, расположить их согласно заданной иерархии и скомпоновать из этого View или экран (рис. 10.15).

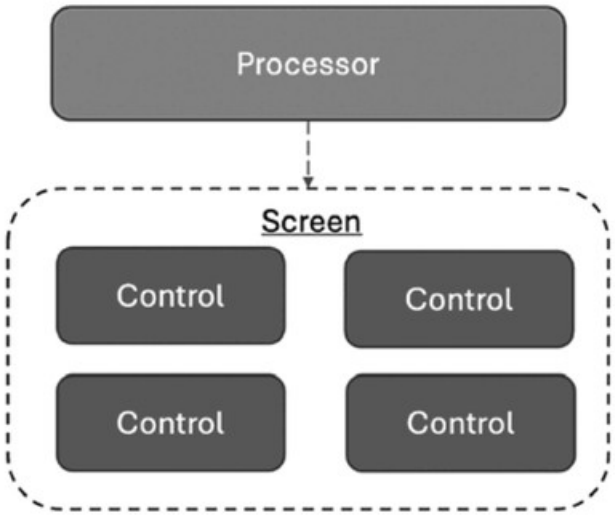


Рис. 10.15. Преобразование JSON процессором

Рассмотрим способы реализации такого процессора в контексте приложения Kotlin Multiplatform (рис. 10.16).

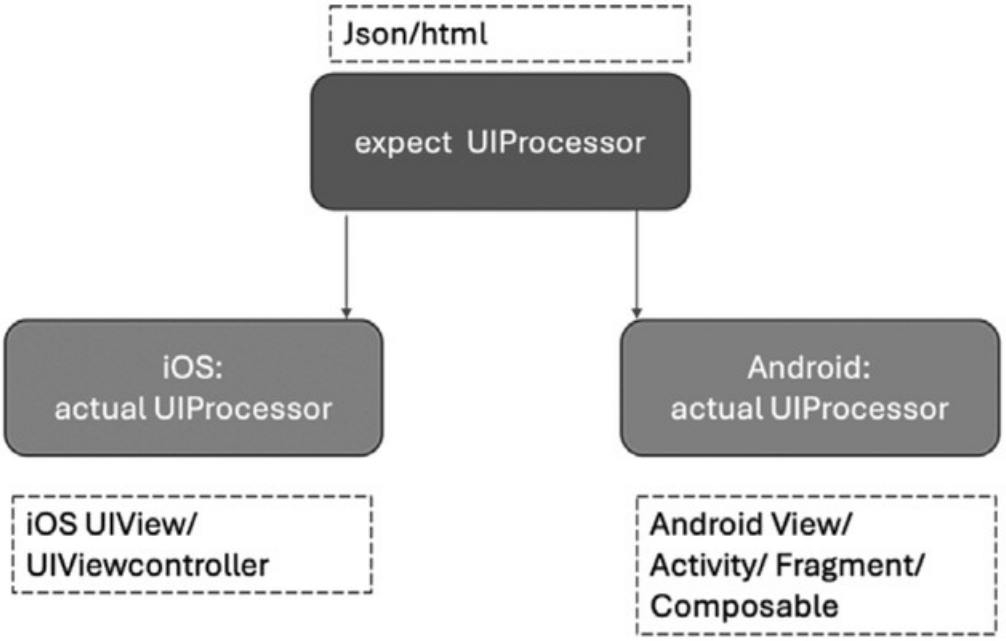


Рис. 10.16. Реализация процессора с помощью expect/actual

На схеме выше приведен возможный вариант реализации процессора с помощью `expect/actual`. Такой подход подойдет, если мы планируем использовать для построения UI стандартные нативные контролы, поддерживаемые SDK. Это будет удобным и в случае, если нам нужно добавить реализуемым компонентам особое поведение. Также имеет смысл рассмотреть использование общего интерфейса, если для создания экрана потребуются компоненты из сторонних библиотек, не поддерживаемые KMP.

Если мы выбираем для реализации UI Compose Multiplatform, достаточно использовать единый процессор в нашем shared-модуле (рис. 10.17).

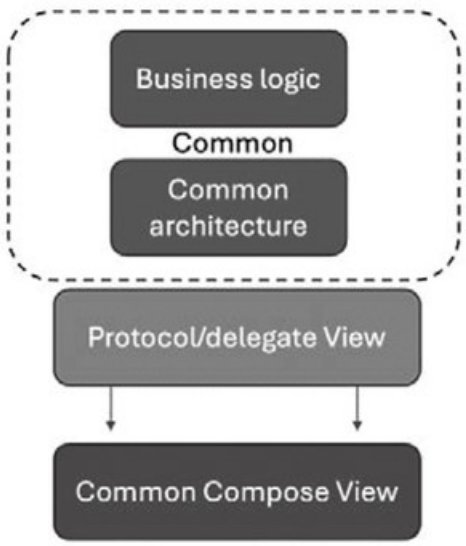


Рис. 10.17. Архитектура приложения с общим UI

Независимо от того, выберем мы единую и общую обработку данных для создания UI или будем делать это отдельно для каждого нативного таргета или в нативном приложении, принцип работы наших процессоров будет одинаковым.

Разберем схему с рис. 10.18.

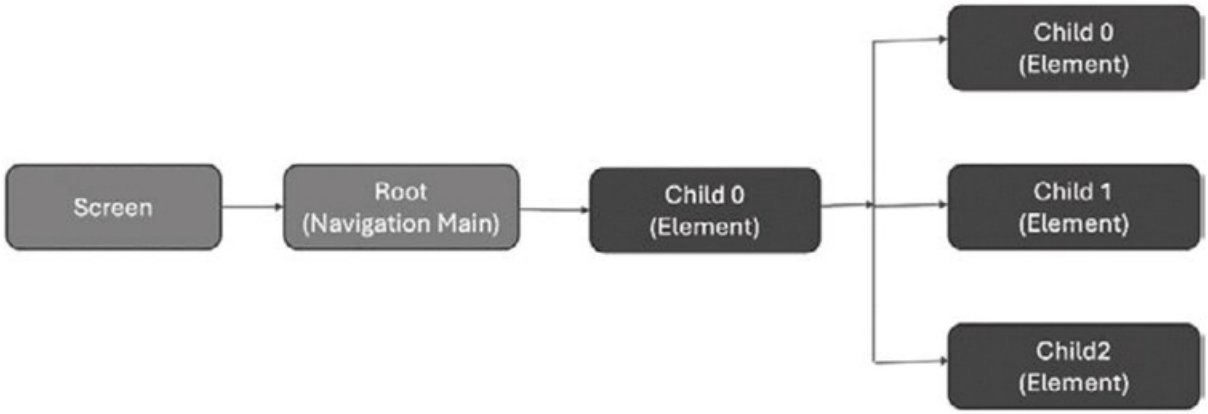


Рис. 10.18. Схема работы рекурсивного процессора

Обычно описание UI в JSON компонуется в древовидную структуру элементов. Каждый узел может быть конечным или иметь потомков. Для того чтобы построить экран, нужно, двигаясь от родительского узла, или корня (Root (Navigation Main)), обойти все узлы и обработать их.

Из технологических шагов процесса обработки и создания экрана складывается следующий алгоритм для SDUI (рис. 10.19).



Рис. 10.19. Порядок шагов обработки элементов

1. Декодирование JSON (JsonDecoder). Сначала необходимо получить исходные модели данных из JSON, сформированного на сервере или в библиотеке.

2. Конвертер (mapper, маппер) экрана. Каждый экран состоит из корневого элемента, внутри которого лежит набор дочерних значений (`subView`). Обработку и преобразование нужно начинать именно с родительского узла, который нужно передавать в маппер в качестве параметра.
3. Конвертер (маппер) элементов. Чтобы обработать каждый элемент правильно, нужно преобразовать его в соответствии с типом.
4. Фабрика элементов. В зависимости от типа элемента вызывается определенный метод или конструктор для создания компонента.

В рамках решения текущей задачи по SDUI мы ограничимся конвертером родительского элемента, который совместит роли декодера, конвертера и процессора. Также добавим конвертер элементов для их правильного создания (рис. 10.20).



Рис. 10.20. Компоненты процессора

В качестве модели данных будем использовать единый SDUI Element. Он может быть как контейнером для дочерних элементов, так и конечным атомарным контролом.

Листинг 10.49. Фрагмент фабрики элементов для создания вертикального списка

```
@Composable
fun SDUIElementVerticalList(element: SDUIElement,
                            modifier: Modifier = Modifier) {
  Column(
    modifier = modifier
      .fillMaxSize()
      .padding(16.dp),
    verticalArrangement = Arrangement.spacedBy(16.dp),
    horizontalAlignment = Alignment.CenterHorizontally
  ) {
    element.children.forEach { child ->
      SDUIElementMapper(child, modifier)
    }
  }
}
```

Код в листинге выше представляет собой Composable-функцию для создания вертикального списка элементов. Для каждого из его дочерних элементов вызывается `SDUIElementMapper`, в котором в зависимости от типа компонента происходит создание Composable нужного типа.

Такой функционал для создания компонентов можно как реализовать вручную, так и оптимизировать с помощью механизмов кодогенерации. Как это сделать, вы сможете узнать подробнее в главе 11.

10.7. Альтернативные решения для общего UI

10.7.1. Библиотека Redwood

Redwood — это библиотека от Square, которая позволяет создавать единый пользовательский интерфейс для различных платформ: Android, iOS и Web. В основе ее работы лежит возможность использовать компилятор Compose без самого Compose UI. Это решение позволяет объявлять общие виджеты пользовательского интерфейса, которые затем реализовывать нативно на каждой отдельной платформе. Результатом будет пользовательский интерфейс на основе встроенного нативного кода под управлением общего кода Redwood. Мы рассматривали такой подход выше, в разделах 10.2.1 и 10.2.2.

Для создания общих элементов в Redwood нужно создать data-класс с описанием виджетов (листинг 10.50).

Листинг 10.50. Общее описание виджета

```
data class TextInput(
  val state: TextFieldState,
  val hint: String = "",
  val onChange: ((TextFieldState) -> Unit)? = null,
)
```

Эти описания называются схемой. На основе схемы Redwood генерирует компонуемый модуль и интерфейс. Каждая платформа привязывает реализацию интерфейса к соответствующему компоненту пользовательского интерфейса (листинг 10.51).

Листинг 10.51. Пример генерации UI на Redwood

```
@Composable
fun TextInput(
        state: TextFieldState,
        hint: String = "",
        onChange: ((TextFieldState) -> Unit)? = null,
        modifier: Modifier = Modifier,
) { … }

interface TextInput<W : Any> : Widget<W> {
        fun state(state: TextFieldState)
        fun hint(hint: String)
        fun onChange(onChange: ((TextFieldState) -> Unit)?)
}
```

Сгенерированные элементы нужно будет подключить к обычному Compose UI. Нативные реализации соединяются внутри механизма благодаря механизму Redwood. Далее из этих общих элементов можно компоновать экраны.

Подробнее вы можете узнать о Redwood из блога Джэка Уортона: https://code.cash.app/native-ui-and-multiplatform-compose-with-redwood.

Redwood — решение очень специфическое и не используется широко. Оно не получило массового распространения из-за стремительного развития Compose Multiplatform, а также из-за закрытой и устаревшей документации.

10.7.2. Библиотека Decompose

Библиотека Decompose — это решение от Аркадия Иванова для управления состоянием и навигацией в приложениях KMP.

Decompose построена вокруг концепции компонентов, которые:

- инкапсулируют бизнес-логику и состояние;
- имеют четкий жизненный цикл;
- могут быть вложенными (иерархия компонентов).

Библиотека включает мощную систему навигации:

- стек-навигацию;
- вкладки (tab navigation);
- панели (split navigation);
- полную поддержку back stack.

Корневым элементом системы является специальный интерфейс RootComponent (листинг 10.52).

Листинг 10.52. Интерфейс RootComponent

```
interface RootComponent {
    val stack: Value<ChildStack<*, Child>>

    sealed interface Child {
        class List(val component: ListComponent) : Child
        class Detail(val component: DetailComponent) : Child
    }
}
```

Компонент содержит декларацию навигационного стека с поддержкой рекурсивной иерархии. Также для удобства работы располагаем в нем декларацию внутреннего sealed-интерфейса Child, чтобы выбрать дочерний компонент для навигации.

Будем использовать два дочерних компонента: для списка новостей и для деталей новости (листинг 10.53).

Листинг 10.53. Описание компонентов

```
interface ListComponent {
    val model: Value<List<NewsItem>>

    fun onNewsItemClicked(item: NewsItem)
}

// Компонент для новости
interface DetailComponent {
    val model: Value<NewsItem>

    fun onBackPressed()
}
```

Реализация каждого компонента может включать в себя всю необходимую для получения контента бизнес-логику (листинг 10.54).

Листинг 10.54. Реализация компонента для деталей новости

```
class DefaultDetailComponent(
    componentContext: ComponentContext,
    newsItem: NewsItem,
    private val onFinished: () -> Unit,
) : DetailComponent, ComponentContext by componentContext {

    override val model: Value<NewsItem> = MutableValue(newsItem)

    override fun onBackPressed() = onFinished()
}
```

Добавим реализацию собственно корневого элемента (листинг 10.55).

Листинг 10.55. Корневой элемент

```
class DefaultRootComponent(
    componentContext: ComponentContext,
) : RootComponent, ComponentContext by componentContext {

// 1 Стек навигации
    private val nav = StackNavigation<Config>()

// 2 Стек дочерней иерархии компонентов
    override val stack: Value<ChildStack<*, RootComponent.Child>> = childStack(
        source = nav,
        serializer = Config.serializer(),
        initialConfiguration = Config.List,
        handleBackButton = true,
        childFactory = ::child,
    )
```

```
// 3 Обработка дочернего компонента
    private fun child(
        config: Config,
        componentContext: ComponentContext
    ): RootComponent.Child = when (config) {/*…*/    }
}
```

Кроме основного стека навигации в качестве параметра для работы с childStack передадим сериализатор типа конфигурации, ссылку на дочернюю фабрику и входную точку.

Для передачи объектов между экранами используем sealed-класс конфигурации (листинг 10.56).

Листинг 10.56. Конфигурационный объект

```
@Serializable
    private sealed interface Config {
        @Serializable
        data object List : Config

        @Serializable
        data class Detail(val newsItem: NewsItem) : Config
    }
```

Для сохранения данных сделаем интерфейсы и его внутренние классы сериализируемыми.

В методе child маршрутизируем создание компонента в зависимости от переданного параметра конфигурации (листинг 10.57).

Листинг 10.57. Реализация метода child

```
private fun child(
    config: Config,
    componentContext: ComponentContext
): RootComponent.Child = when (config) {

    Config.List -> RootComponent.Child.List(
        DefaultListComponent(
            componentContext = componentContext,
            newsItemClicked = { newsItem ->
// Вызов перехода, передаем элемент из события клика
nav.pushNew(Config.Detail(newsItem)) }
        )
    )

    is Config.Detail -> RootComponent.Child.Detail(
        DefaultDetailComponent(
            componentContext = componentContext,
            newsItem = config.newsItem,
            onFinished = { nav.pop() },
        )
    )
}
```

При конфигурации компонента в качестве параметров используем контекст компонента, контент и/или обработчики событий. В качестве события передадим вызов перехода к следующему экрану (`pushNew`) или возврат назад. В метод `pushNew` возьмем объект конфигурации, соответствующий точке перехода.

Входной точкой для подключения к нативным приложениям должен быть Composable. Создадим специальный `RootContent`, в качестве параметра используем `RootComponent` (листинг 10.58).

Листинг 10.58. Composable для контента корневого элемента

```
@Composable
fun RootContent(
    component: RootComponent,
    modifier: Modifier = Modifier,
) {
    Children(
        stack = component.stack,
        modifier = modifier,
        animation = stackAnimation(slide()),
    ) {
        when (val child = it.instance) {
            is RootComponent.Child.Detail -> DetailContent(
                component = child.component,
                modifier = Modifier.fillMaxSize(),
            )

            is RootComponent.Child.List -> ListContent(
                component = child.component,
                modifier = Modifier.fillMaxSize(),
viewModel = KoinDIFactory.resolve(NewsViewModel::class.java)

            )
        }
    }
}
```

Во внутренний блок `Children` добавим ссылку на стек навигации корневого компонента. Внутри `Children` реализуем обработку текущего элемента стека, который нужно отрисовать. В зависимости от типа элемента вызовем отрисовку нужного Composable. Это может быть тот же Composable, который мы использовали для создания списка новостей. Для поддержания логики работы Decompose передадим нужный дочерний компонент. Реализуем контент `ListContent` примерно так (листинг 10.59):

Листинг 10.59. Реализация контента с передачей компонента

```
@Composable
fun ListContent(
        component: ListComponent,
        modifier: Modifier = Modifier,
        viewModel: NewsViewModel,
```

```
) {
    val state by component.model.subscribeAsState()

    NewsListScreen(viewModel = viewModel,
            onClick = {component.onNewsItemClicked(it)})
}
```

В событии `onClick` вызовем событие `onNewsItemClicked` компонента.

Получим рабочее решение.

Подробнее о работе с Decompose вы можете узнать здесь:

- https://github.com/arkivanov/Decompose
- https://arkivanov.github.io/Decompose

ПОДВЕДЕМ ИТОГИ

- ✓ Есть много способов работы с нативным UI как с общим, но самый эффективный и быстрый — Compose Multiplatform.
- ✓ Compose Multiplatform позволяет создавать UI так же просто, как Jetpack Compose, и реализовывать его сразу под несколько таргетов.
- ✓ Для реализации навигации можно использовать такие библиотеки, как PreCompose, которые позволяют управлять состоянием приложения так же, как Android Jetpack Navigation. Также хорошим решением является и Decompose.

ЧАСТЬ II

ГЛАВА 11

Полезные инструменты для Kotlin Multiplatform

В этой главе рассмотрим:

- способы оптимизации кода в KMP;
- аннотации и KSP для генерации кода;
- плагины компилятора для решения простых задач;
- ИИ-помощники для ускорения и улучшения разработки;
- полезные библиотеки.

Кодогенерация — один из способов оптимизации разработки приложений. С ее помощью можно сократить количество кода, которое пишет разработчик, автоматизировать создание модулей тестов, упростить архитектуру приложения и многое другое. Механизмы кодогенерации можно применять ко всем слоям кода как нативного, так и кросс-платформенного приложения — вопрос в подборе правильного инструмента для задачи.

11.1. Аннотации и Kotlin Symbol Processing

Сбор данных для анализа или кодогенерации можно реализовать через метапрограммирование. Этот механизм позволяет получать метаинформацию во время работы или компиляции кода. Одним из инструментов метапрограммирования является обработка (процессинг) аннотаций Kotlin Symbol Processing или Annotation Processing. Аннотации в данном случае выступают как маркеры для процессора, указывающие на то, какой код надо анализировать и по каким правилам (листинг 11.1).

Листинг 11.1. Примеры аннотаций

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.SOURCE)
annotation class Adapter(
    val holders: Array<KClass<*>>,
    val bindType: KClass<*>
)
```

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.SOURCE)
annotation class BindVH(
    val bindType: KClass<*>
)

@Retention(AnnotationRetention.SOURCE)
@Target(AnnotationTarget.FUNCTION)
annotation class BindSetup

@Target(AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.SOURCE)
annotation class BindLayout

@Retention(AnnotationRetention.SOURCE)
@Target(AnnotationTarget.FUNCTION)
annotation class BindView
```

Технология Kotlin Symbol Processing была представлена Google в феврале 2021 года как потенциальная кросс-платформенная замена KAPT (Kotlin Annotation Processing Tool). KAPT — мощный инструмент для анализа и генерации кода, в старых версиях Dagger и Room для Android используется именно он. Но для кросс-платформы он не подходит из-за тесной зависимости от Java и JVM: прежде чем проанализировать и сгенерировать код, KAPT создает Java-заглушки для решения задачи. Затем они компилируются и обрабатываются. В качестве результата мы получаем также файлы для JVM (рис. 11.1).

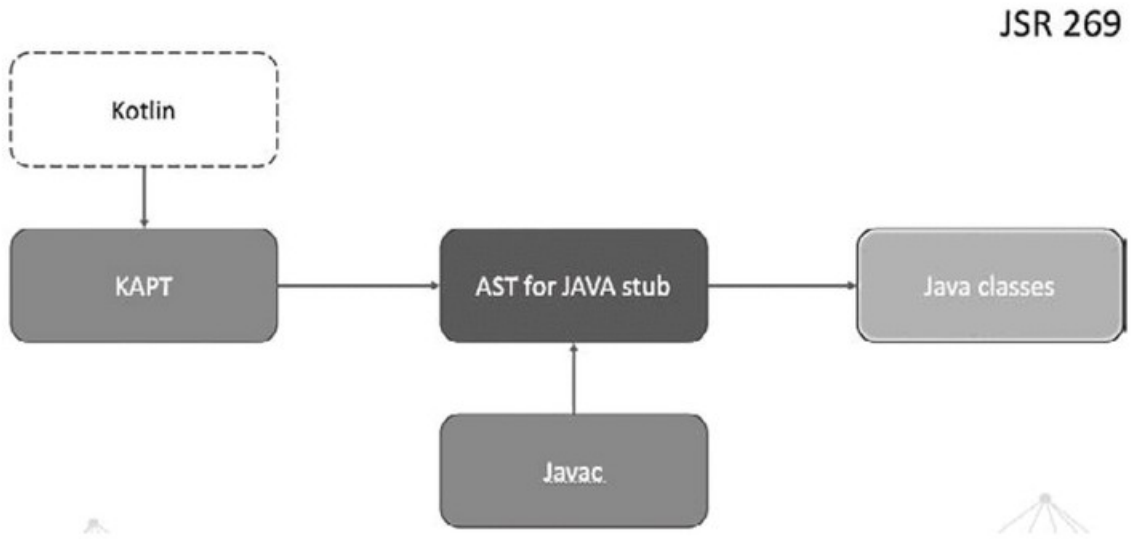


Рис. 11.1. Схема работы KAPT

В отличие от KAPT, в KSP нет никакой генерации Java-заглушек. Процессор работает с AST (Abstract Syntax Tree, абстрактное синтаксическое дерево) Kotlin напрямую, что позволяет сразу генерировать Kotlin-код, причем именно тот, который мы будем использовать в приложении. За счет этого работать с KSP

получается быстрее, эффективнее и чище. С 2023 года KSP — официально рекомендуемое Google решение для работы с аннотациями для KMP и Android.

Для создания своего процессора аннотации нужно реализовать интерфейс SymbolProcessor (листинг 11.2).

Листинг 11.2. Интерфейс SymbolProcessor

```
interface SymbolProcessor {

  fun process(resolver: Resolver): List<KSAnnotated>

  fun finish() {}

  fun onError() {}
}

interface SymbolProcessorProvider {

  fun create(environment: SymbolProcessorEnvironment): SymbolProcessor
}
```

Для начала работы нам потребуется зарегистрировать провайдер процессора как ресурс (рис. 11.2).

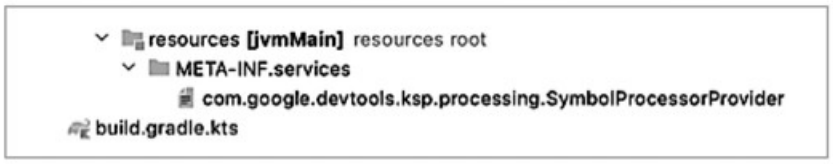


Рис. 11.2. Регистрация SymbolProcessorProvider

Внутри файла регистрации нужно указать все интерфейсы типа `SymbolProcessorProvider` проекта (листинг 11.3).

Листинг 11.3. Декларация интерфейса SymbolProcessorProvider

```
// Декларируем все ProcessorProvider
com.azharkova.processor.core.UsecaseProcessorProvider
```

Анализ данных и обработка вызываются в основном методе `process` с помощью компонента `Resolver`, который работает со всеми метаединицами типа `KSAnnotated` (листинг 11.4).

Листинг 11.4. Пример работы процессора и Resolver

```
class AdapterProcessor constructor(val env: SymbolProcessorEnvironment) :
SymbolProcessor {

  val visitor = CustomVisitor()

  override fun process(resolver: Resolver): List<KSAnnotated> {
```

```
    // 1 Получаем адаптеры
    val adapters = getAdapters(resolver)

    /**...*/
    // 2 Генерируем класс
    generateImplClass(data, codeGenerator)
    return emptyList()
  }

  private fun getAdapters(resolver: Resolver): Sequence<KSClassDeclaration> {
    // Фильтруем все классы с аннотацией адаптер
    return resolver.getSymbolsWithAnnotation(Adapter::class.java.name)
        .filterIsInstance<KSClassDeclaration>()
        .distinct()
  }
}
```

Анализ можно также проводить с помощью специального класса-визитера `KSVisitorVoid` (листинг 11.5).

Листинг 11.5. Пример визитера

```
class CustomVisitor : KSVisitorVoid() {

  override fun visitClassDeclaration(
    classDeclaration: KSClassDeclaration,
    data: Unit
  ) {
    classDeclaration.getDeclaredFunctions().map { it.accept(this, Unit) }
  }

  override fun visitFunctionDeclaration(
    function: KSFunctionDeclaration,
    data: Unit
  ) {
    functions.add(function)
  }

  override fun visitFile(file: KSFile, data: Unit) {
    file.declarations.map { it.accept(this, Unit) }
  }
}
```

Методы этого класса позволяют собирать метаданные в зависимости от их типа: `KSClassDeclaration`, `KSFunctionDeclaration`, `KSFile`.

Мы рассматривали, как подключать KSP к проекту, когда знакомились с Room и другими библиотеками, работающими с кодогенерацией (листинг 11.6).

Листинг 11.6. Подключение плагина и зависимостей

```
plugins {
  //...
  alias(libs.plugins.ksp) apply false
```

```
}
/*build.gradle.kts shared*/
plugins {
  //...
  id("com.google.devtools.ksp")
}
```

Рекомендуется вынести процессор аннотаций на KSP в отдельный модуль и затем подключить его к shared (рис. 11.3).

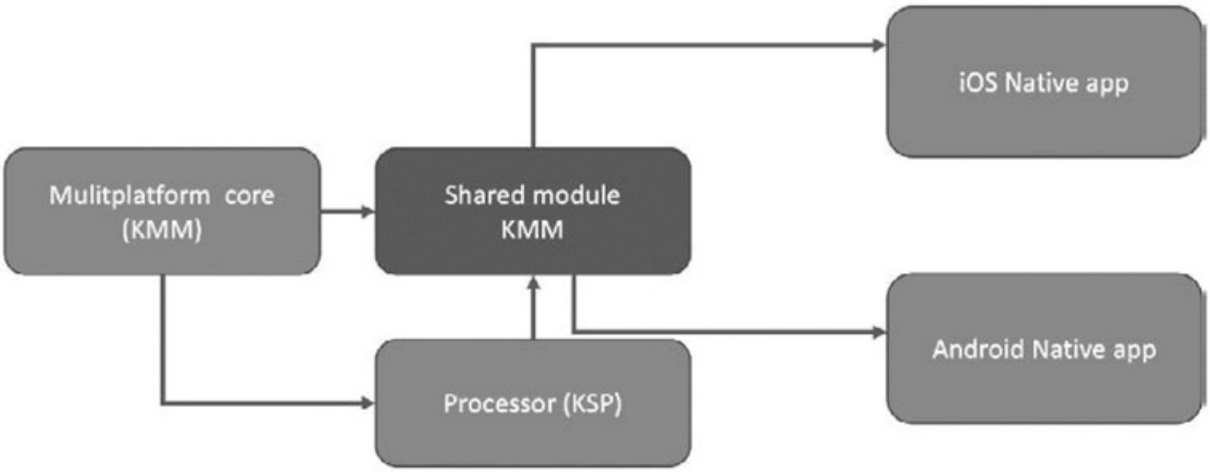


Рис. 11.3. Подключение KSP модуля

Часть кода, который нужен и для shared, и для работы процессора, лучше тоже вынести отдельно и подключить как зависимость в оба модуля (shared и processor).

В shared-модуль тоже нужно добавить обработку модуля с процессором для каждого таргета (листинг 11.7).

Листинг 11.7. Подключение модуля с процессором к shared

```
sourceSets {
    commonMain.dependencies {
            implementation(project(":processor"))
    }
}
// Подключение модуля с генерацией
dependencies {
    add("kspCommonMainMetadata", project(":processor"))
    add("kspIosX64", project(":processor"))
    add("kspIosSimulatorArm64", project(":processor"))
    add("kspIosArm64", project(":processor"))
    add("kspAndroid", project(":processor"))
}
```

Попробуем разобрать простой пример. Представим, что мы решили создать свой генератор `Usecase` (рис. 11.4).

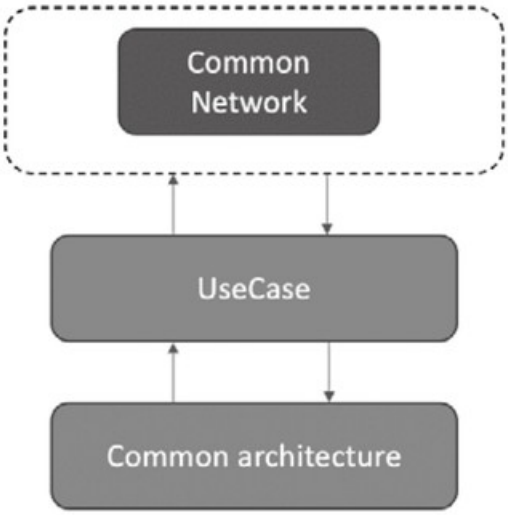


Рис. 11.4. Архитектура с UseCase

Такой `UseCase` мы использовали для прямого обращения к сервису с поддержкой обработки ошибок от корутин (главы 2, 4, 6).

В листинге 11.8 приведен пример использования аннотаций для `Usecase`.

Листинг 11.8. Аннотация для UseCase

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.SOURCE)
annotation class UseCase(
    val repo: KClass<*>,
    val request: RequestType
)

// Использование
@UseCase(repo = NewsApi::class, request = RequestType.NEWS)
interface SimpleNewsLoadCase
```

В качестве параметров передадим ссылку на API для вызова и перечисление `enum` для определения метода запроса.

Подготовим модель для сбора информации (листинг 11.9).

Листинг 11.9. Модель данных для будущего Usecase

```
data class UsecaseData (
    var name: String = "",
    var packageName: String = "",
    var imports: List<String> = emptyList(),
    var requestClazz: ClazzInfo? = null,
    var apiName: String? = null,
    var methodName: String? = null,
    val paramType: ReturnsTypeData? = null,
    val returnsTypeData: ReturnsTypeData? = null
)
```

Для генерации потребуются: название `UseCase`, имя пакета, импорты, имя API-класса, название метода, тип ответа.

Сначала нам нужно отобрать все классы, отмеченные аннотацией `@UseCase`, и получить из них информацию через `KClassDeclaration`.

Подготовим метод расширения для преобразования `KClassDeclaration` в данные для `UseCase` (листинг 11.10).

Листинг 11.10. Сбор информации

```
fun KSClassDeclaration.toUsecaseData(resolver: Resolver): UsecaseData? {

// Получение аннотации
  return getUsecaseAnnotation(this)?.let { annotation ->
    // 1 Имя репозитория
    val api = getParamValue(annotation, "repo")
    // 2 Название метода
    val method = getParamValueSimple(annotation, "request")
    // 3 Ссылка на класс, содержащий функцию
    val requestClazz = getFunction(
        resolver,
        BindRequest::class.java.name,
        method.toString(),
        logger
    ).groupBy { it.closestClassDeclaration()!! }
     .map { (classDesc, funcS) ->
         classDesc.toClazzData(logger, resolver, funcS)
     }.firstOrNull { it.name == api.toName() }

    // Собранные данные
    UsecaseData(
        simpleName.asString(),
        packageName.asString(),
        requestClazz = requestClazz,
        apiName = api?.declaration?.simpleName?.asString(),
        methodName = method?.toString(),
        paramType = paramType,
        returnsTypeData = returnType
    )
  }
}
```

Нам нужно получить доступ и к самой аннотации, чтобы взять из нее параметры класса Api/сетевого клиента, и к методу, который мы будем запрашивать. Нам также потребуется ссылка на класс, которому принадлежит метод запроса. Остальные параметры мы возьмем из KClassDeclaration.

Собранные данные мы будем использовать для шаблона с кодом. Шаблон можно подготовить как в виде строки текста, так и с помощью других инструментов, например Kotlin Poet.

Для генерации данных запустим задачу на сборку (листинг 11.11).

Листинг 11.11. Запуск Gradle-задач на генерацию данных

```
./gradlew kspKotlin  # Только генерация KSP
./gradlew build      # Полная сборка с генерацией
```

Сгенерированный результат мы сможем найти в каталоге ksp внутри build (рис. 11.5).

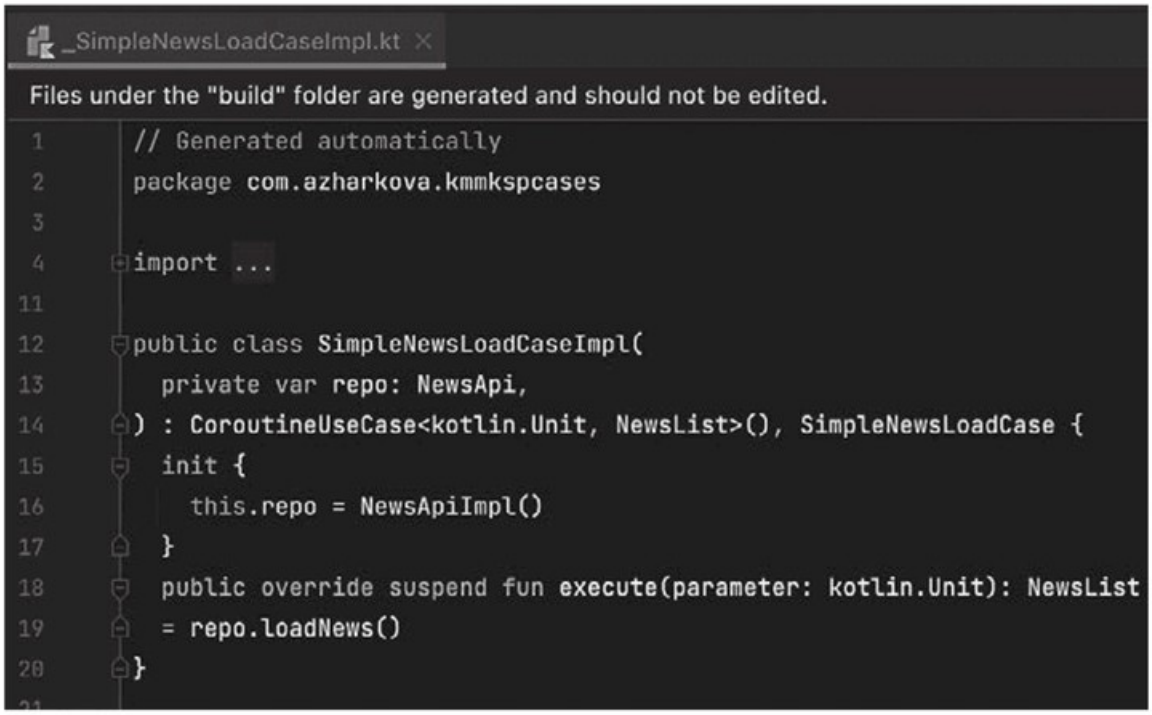


Рис. 11.5. Результат генерации

Подробнее о KSP читайте в документации https://github.com/google/ksp.

Основной недостаток KSP состоит в том, что эта технология позволяет создавать только новые файлы, но не менять существующие.

11.2. Плагины компилятора

Еще одним способом решения задачи через метапрограммирование является использование плагинов компилятора Kotlin. Compose, Room, Mochi, Kotlinx. Serialization — эти технологии (и не только они) используют плагины компилятора. KSP не только является плагином, но и помогает писать легковесные плагины компилятора, которые работают немного на другом уровне, чем процессоры аннотаций (рис. 11.6).

Если KSP может создавать только новые файлы, то плагины компилятора позволяют менять существующий код.

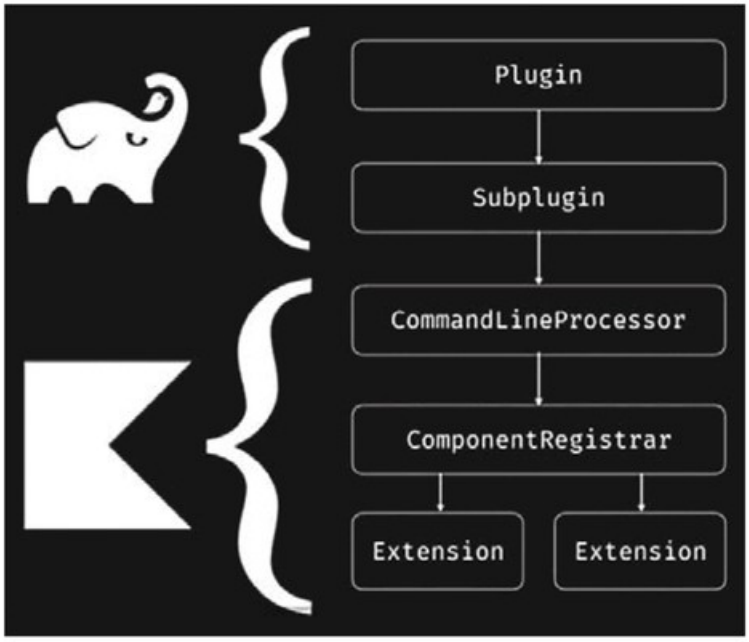


Рис. 11.6. Плагин компилятора

KCP (KotlinCompilerPlugin) имеет прямой доступ к абстрактному синтаксическому дереву и меняет его узлы в формате IR (промежуточное представление). IR используется компиляторами для оптимизации и преобразования исходного кода перед генерацией байт-кода. Его использование позволяет унифицировать обработку кода разными VM.

IR представляет собой иерархическую структуру узлов кода (листинг 11.12).

Листинг 11.12. Пример IR (промежуточного представления)

```
FUN name:test visibility:public modality:FINAL <> () returnType:kotlin.Unit
              [suspend]
  BLOCK_BODY
    CALL 'public final fun println (message: kotlin.Int): kotlin.Unit [inline]
    declared in kotlin.io.ConsoleKit' type=kotlin.Unit origin=null
    message: CALL 'public final fun returnsInt(): kotlin.Int [suspend]
    declared in <root>.TestKit' type=kotlin.Int origin=null
```

Плагин компилятора состоит из следующих компонентов (рис. 11.7):

- `KcpGradlePlugin` — плагин `KotlinCompilerPluginSupportPlugin`, используется для описания плагина и таргетов.

  ВНИМАНИЕ! При работе с кросс-платформенным кодом потребуется прописать несколько артефактов, особенно если нужно поддержать разный код.

- `PluginComponentRegistrar` — класс для регистрации самого плагина компилятора.
- `PluginGenerationExtension` — класс для регистрации расширения для плагина генерации.

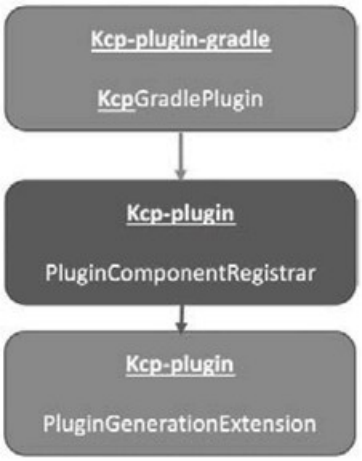


Рис. 11.7. Структура плагина

Рассмотрим простой пример: напишем плагин, который будет выводить модель данных с полями (листинг 11.13).

Листинг 11.13. Пример работы плагина

```
@AutoToString
data class User(val name: String, val age: Int)

fun main() {
    val user = User("Alice", 25)
    println(user.toString()) // Должно вывести: User(name=Alice, age=25)
}
```

Начнем с установки зависимостей (листинг 11.14).

Листинг 11.14. Зависимости для плагина компилятора

```
plugins {
    id("kotlin-multiplatform") version "2.0.0"
    id("org.jetbrains.kotlin.plugin.compiler") version "2.0.0"
}

dependencies {
    implementation(kotlin("compiler-embeddable"))
    implementation("org.jetbrains.kotlin:kotlin-compiler-plugin-api:2.0.0")
}
```

Пропишем зависимости для плагина компилятора в `build.gradle.kts`. Используем версию с поддержкой K2.

В качестве маркера для применения нашего плагина к коду будем использовать аннотацию (листинг 11.15).

Листинг 11.15. Аннотация для примера

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.SOURCE)
annotation class AutoToString
```

Напишем расширение для внедрения кода в существующий метод `toString` (листинг 11.16). Для трансформации отдельных фрагментов кода будем использовать `IrElementTransformerVoid`. Если бы нам требовалось более плотно работать со всем кодом класса, то мы бы взяли `ClassLoweringPass`.

Листинг 11.16. Пример реализации плагина

```
class AutoToStringTransformer(
        private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {

  override fun visitClass(declaration: IrClass): IrStatement {

    // Применяем для всех классов AutoToString
    if (declaration.hasAnnotation(FqName("AutoToString"))) {
      addToStringMethod(declaration)
    }
    return super.visitClass(declaration)
  }

  private fun addToStringMethod(irClass: IrClass) {
    val irFactory = pluginContext.irFactory
// Внедряем код в метод toString
    val toStringFunction = irFactory.buildFun {
      name = Name.identifier("toString")
      returnType = pluginContext.irBuiltIns.stringType
      modality = Modality.OPEN
    }.apply {
      parent = irClass
// Заменяем body функции
      body = IrBlockBodyImpl(
          startOffset, endOffset,
          listOf(
// Возврат данных
              irReturn(irConcat().apply {
                // Начинаем с названия класса и скобки
                arguments.add(irString("${irClass.name.identifier}("))
                // Для каждого из полей
                irClass.declarations
                    .filterIsInstance<IrProperty>()
                    .forEachIndexed { index, property ->
                      if (index > 0) arguments.add(irString(", "))
                      // Добавляем имя поля
                      arguments.add(irString("${property.name.asString()}="))
                      // Выводим значение поля
                      arguments.add(irGet(property.backingField!!))
                    }
                arguments.add(irString(")"))
              })
          )
      )
```

```
    }
// Добавляем функцию классу
    irClass.declarations.add(toStringFunction)
  }
}
```

Наш код должен влиять на все классы, которые будут отмечены аннотацией `@AutoToString`. Поэтому нужно создать класс, реализующий `IrElementTransformerVoid`, и переопределить метод `visitClass`. Этот метод в качестве параметра получает `IrClass`. Нужно отделить все классы с аннотацией `@AutoToString` и переопределить для них метод `toString`.

В методе `addToString` создадим перегрузку функции. Из контекста плагина получим фабрику для работы с промежуточными представлениями. С ее помощью создадим функцию и пропишем у нее `body IrBlockBody`. Затем отфильтруем все свойства класса и для каждого соберем данные: имя поля и значение. Готовое расширение зарегистрируем как `IrGenerationExtension` (листинг 11.17).

Листинг 11.17. Регистрация расширения

```
import org.jetbrains.kotlin.backend.common.extensions.*
import org.jetbrains.kotlin.ir.declarations.*
import org.jetbrains.kotlin.ir.util.*

class AutoToStringIrExtension : IrGenerationExtension {
    override fun generate(moduleFragment: IrModuleFragment,
                          pluginContext: IrPluginContext) {
// Вызываем наш класс-трансформер и передаем контекст плагина
        moduleFragment.transform(AutoToStringTransformer(pluginContext), null)
    }
}
```

При создании расширения передадим в конструктор контекст плагина. Затем надо зарегистрировать и сам плагин компилятора (листинг 11.18).

Листинг 11.18. Регистрация плагина

```
import org.jetbrains.kotlin.compiler.plugin.*
import org.jetbrains.kotlin.config.CompilerConfiguration

class AutoToStringPluginRegistrar : CompilerPluginRegistrar() {
    override val supportsK2: Boolean = true

    override fun ExtensionStorage.registerExtensions(configuration:
                                  CompilerConfiguration) {
        IrGenerationExtension.registerExtension(AutoToStringIrExtension())
    }
}
```

После этого проверим работу кода.

Аналогичным способом мы можем решать другие задачи:

- генерация архитектурных компонентов (например, useCase);
- генерация DI;
- оптимизация UI.

Эти задачи объемные, и мы не будем сейчас на этом останавливаться. К тому же цель этой главы — просто познакомить вас с принципами работы и реализации плагинов компилятора. Подробнее о работе с плагинами компилятора Kotlin вы можете узнать из этой серии статей: https://blog.bnorm.dev/writing-your-second-compiler-plugin-part-1.

Мы получили легковесное решение атомарной и простой задачи. При создании более сложных и серьезных плагинов вы можете столкнуться со следующими проблемами:

- недостаточная документация в открытом доступе. Та, которая есть, в основном касается внутренних разработок компаний-контрибьюторов;
- нестабильное API.

11.3. ИИ-помощники для кросс-платформенной разработки

Использование искусственного интеллекта (AI, ИИ) в качестве помощника в разработке приложений, в том числе и мобильных, становится трендом последнего времени. Обученные нейронные сети позволяют анализировать код, искать ошибки и проблемы, покрывать тестами, генерировать документацию, а также решать другие кодовые задачи.

Таких сервисов в настоящее время довольно много:

- ChatGPT от OpenAI;
- Claude Sonnette;
- Gemini (Google);
- YandexGPT/Yandex LLM от Yandex;
- GigaCode от Сбер;
- Deepseek.

Есть и открытые LLM, которые можно загрузить с сайта https://huggingface.co и развернуть локально на своей рабочей машине или во внутренней сети.

Готовые сервисы доступны не только на специальных веб-сайтах. Некоторые из них реализованы в виде готовых плагинов для IDE Android Studio. В качестве помощников-плагинов мы рассмотрим ассистентов Gemini и GigaCode.

11.3.1. Gemini

На Google I/O 2024 компания представила обновленную версию SDK Gemini (бывший Bard). Основной акцент был сделан на использование этого функционала через встроенный плагин для IDE Android Studio.

Начнем с установки и настройки Gemini. Так как с версии Android Koala Gemini встроен в IDE, нам достаточно авторизоваться под своей учетной записью Google (рис. 11.8).

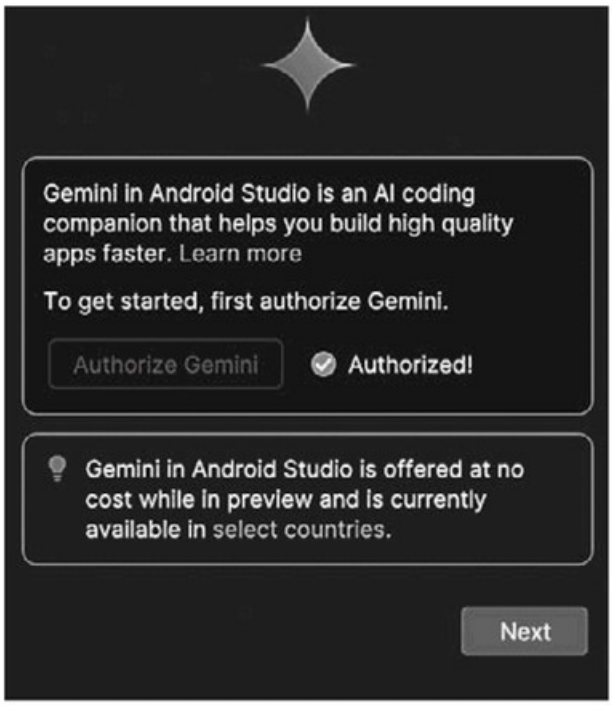


Рис. 11.8. Авторизация в Gemini

> **ВНИМАНИЕ!** Google ограничивает зону доступности Gemini определенными странами. Если вы территориально не попадаете в этот список, то перед открытием веб-версии Gemini или IDE воспользуйтесь доступным для вас механизмом смены местоположения.

Политика конфиденциальности предупреждает, что когда вы используете Gemini в Android Studio, Google собирает ваши запросы, связанный с ними

код, сгенерированные выходные данные, информацию об использовании соответствующих функций и ваши отзывы. Google использует эти данные для предоставления, улучшения и разработки продуктов и сервисов и для развития технологий машинного обучения, включая корпоративные продукты Google, например Google Cloud.



Рис. 11.9. Политика конфиденциальности Google

Для генерации ответов Gemini может ориентироваться на контекст всех ваших проектов либо одного конкретного проекта (рис. 11.10).

Это означает, что код, разрешенный к ознакомлению, будет изучаться ИИ. Возможно, какие-то образцы будут фиксироваться на удаленном сервере. Вы также можете запретить брать контекст из вашего кода, чтобы избежать утечки данных.

После авторизации в сервисе Gemini предлагает несколько готовых запросов для работы (рис. 11.11).

Можно также вызвать плагин из контекстного меню и применить его к выделенному коду (рис. 11.12).

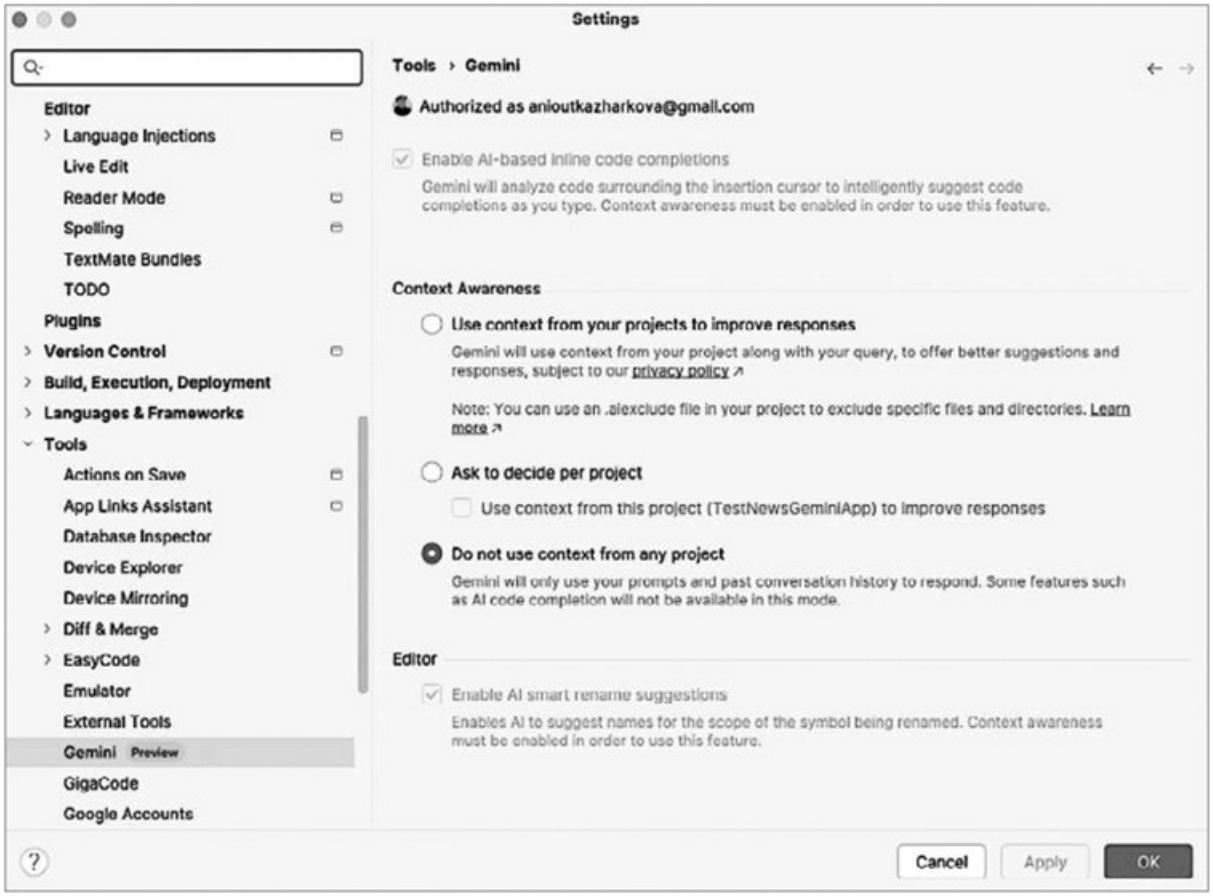


Рис. 11.10. Настройки плагина Gemini



Рис. 11.11. Готовые промты Gemini

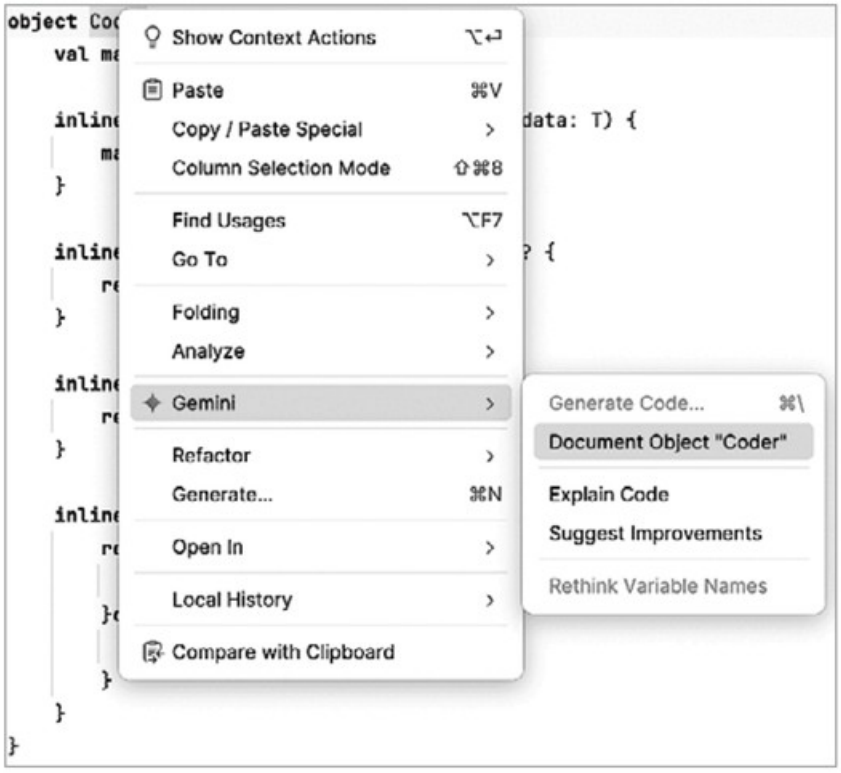


Рис. 11.12. Работа с Gemini через контекстное меню

После выбора функции промт с кодом вызывается в поле чата с Gemini (рис. 11.13).



Рис. 11.13. Применение промта к коду

Рассмотрим возможности помощников ИИ (AI) на примере задачи с SDUI для Compose из главы 10. Реализация парсера и конвертера из JSON в компоненты описания экрана — рутинная и хлопотная задача, которая требует времени. Попробуем переложить ее выполнение на искусственный интеллект. Давайте сразу посмотрим на финальный результат нашей задачи. С помощью ИИ-помощника из нашего исходного JSON мы сгенерируем модели элементов для конструктора SDUI (листинг 11.19).

Листинг 11.19. Модели данных для SDUI

```
// Enum для типов элементов
enum class SDUIElementType {
    SCAFFOLD,
    VERTICAL_LIST,
    ROW,
    TEXT,
    IMAGE,
    CENTER_ALIGNED_TOP_APP_BAR
}

// Базовая модель элемента
@Serializable
data class SDUIElement(
    val type: SDUIElementType,
    val value: String? = null,
    @SerialName("top_bar") val topBar: List<SDUIElement>? = null,
    val children: List<SDUIElement>? = null
)
```

Каждый из элементов может содержать внутренние дочерние элементы. Выделенное свойство для заголовка topBar является особенностью конкретного JSON.

Мы также получим Composable для каждого типа элемента (листинг 11.20).

Листинг 11.20. Примеры Composable для SDUI элементов

```
// Composable для текста
@Composable
fun SDUIElementText(element: SDUIElement,
    modifier: Modifier = Modifier) {
    Text(
        text = element.value ?: "",
        style = MaterialTheme.typography.body1,
        modifier = modifier.padding(16.dp)
    )
}

// Composable для изображения
@Composable
fun SDUIElementImage(element: SDUIElement,
    modifier: Modifier = Modifier) {
    SDUIImage(
        url = element.value ?: "",
```

```
            modifier = modifier.size(100.dp)
        }
}
```

Конвертер элементов в листинге тоже полностью сгенерирован. Это Composable-функция, в которой в зависимости от элемента и его типа вызывается нужный метод создания (листинг 11.21).

Листинг 11.21. Пример конвертера с фабрикой

```
@Composable
fun SDUIElementMapper(element: SDUIElement,
    modifier: Modifier = Modifier) {
    when (element.type) {
        SDUIElementType.SCAFFOLD -> {
            SDUIElementScaffold(element)
        }
        SDUIElementType.VERTICAL_LIST -> {
            SDUIElementVerticalList(element)
        }
        SDUIElementType.ROW -> {
            SDUIElementRow(element)
        }
        SDUIElementType.TEXT -> {
            SDUIElementText(element)
        }
        SDUIElementType.IMAGE -> {
            SDUIElementImage(element)
        }
        else -> {}
    }
}
```

Еще одним важным компонентом для решения задачи, который сгенерирует нам Gemini, является родительский конвертер (листинг 11.22). Его роль — рекурсивно обойти все элементы внутри родительской модели и вызвать для каждого из них фабрику.

Листинг 11.22. Родительский конвертер

```
@Composable
fun SDUIElementParentMapper(json: String) {
    val parent: SDUIElementParent = Json.decodeFromString(json)
    parent.data.forEach { element ->
        SDUIElementMapper(element)
    }
}
```

В финальной версии на вход будет подаваться строка JSON, которая затем десериализуется в родительский элемент.

На первый взгляд все эти фрагменты кода выглядят неплохо и кажутся исчерпывающими для решения задачи SDUI для UI на Compose Multiplatform.

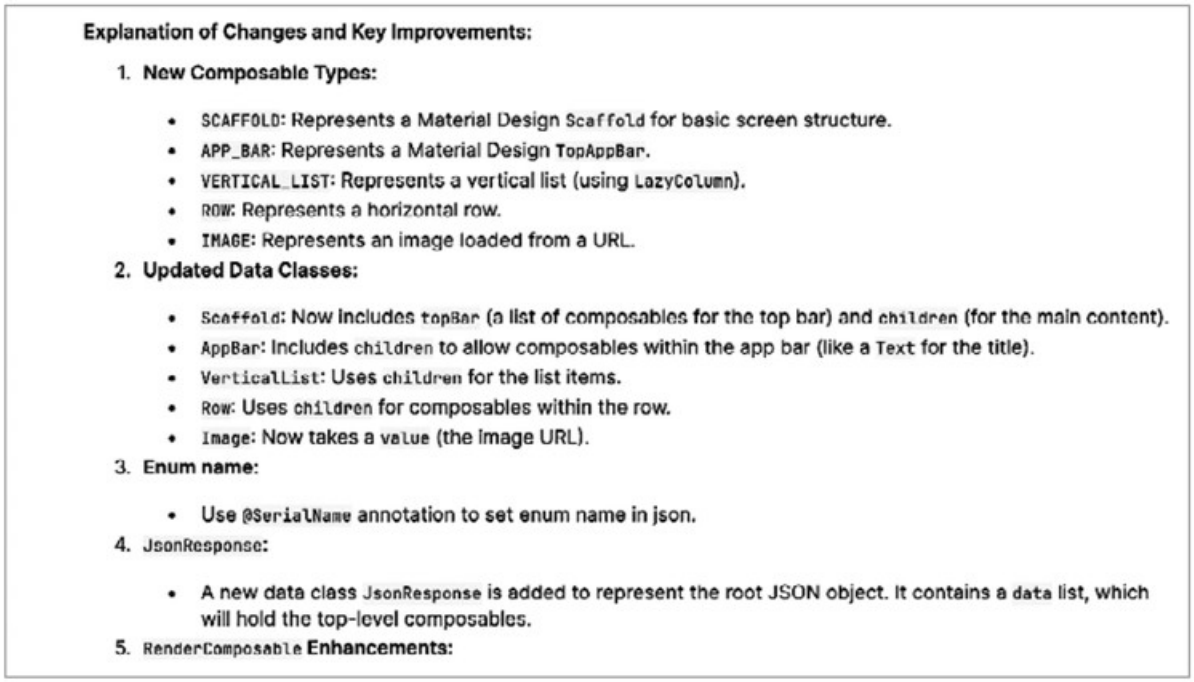
Проблема в том, что мы получили эти данные не с первого раза, и пришлось потратить время на подбор промта и уточнение запросов.

Давайте разберем, с какими проблемами генерации кода можно столкнуться и какие нюансы нужно учитывать.

1. Ваш запрос должен быть максимально детализированным. Например, для создания конвертера из JSON в Composable нужно поставить саму задачу, указать, что: а) необходима поддержка всех нужных импортов, Compose Multiplatform, код-стайла Kotlin; б) нельзя использовать принудительную распаковку опционалов; в) следует оптимизировать код с помощью перечислений. Итоговый запрос показан в листинге ниже.

   ```
   Create a mapper from JSON to composable functions. Add all needed imports,
   add support to Compose Multiplatform, use kotlin code style, don't use force
   unwrap, use enums
   ```

   Вы получите подробный ответ, даже с объяснениями (рис. 11.14).



Explanation of Changes and Key Improvements:

1. **New Composable Types:**
   - `SCAFFOLD`: Represents a Material Design `Scaffold` for basic screen structure.
   - `APP_BAR`: Represents a Material Design `TopAppBar`.
   - `VERTICAL_LIST`: Represents a vertical list (using `LazyColumn`).
   - `ROW`: Represents a horizontal row.
   - `IMAGE`: Represents an image loaded from a URL.
2. **Updated Data Classes:**
   - `Scaffold`: Now includes `topBar` (a list of composables for the top bar) and `children` (for the main content).
   - `AppBar`: Includes `children` to allow composables within the app bar (like a `Text` for the title).
   - `VerticalList`: Uses `children` for the list items.
   - `Row`: Uses `children` for composables within the row.
   - `Image`: Now takes a `value` (the image URL).
3. **Enum name:**
   - Use `@SerialName` annotation to set enum name in json.
4. `JsonResponse`:
   - A new data class `JsonResponse` is added to represent the root JSON object. It contains a `data` list, which will hold the top-level composables.
5. `RenderComposable` **Enhancements:**

Рис. 11.14. Пример ответа от нейросети с объяснениями

   Особенность работы с ИИ-помощниками состоит в том, что ответ на один и тот же вопрос может быть каждый раз разным. Нейросети накапливают опыт, обучаются и генерируют следующий свой ответ как предположение или предсказание, похожее на логический вывод, на основе накопленной информации.

2. Если нейросеть чего-то не знает, ее предположение может оказаться неверным. Например, Gemini может выдумывать несуществующие функции и элементы (листинг 11.23).

Листинг 11.23. Пример AppBar с ошибкой

```
Scaffold(
  topBar = {
    AppBar(
      title = {
        Text(
          text = data.data.first().top_bar!!
            .first().children!!.first().value!!,
          modifier = Modifier.fillMaxSize(),
          textAlign = TextAlign.Center
        )
      }
    )
  }
)
```

В приведенном выше фрагменте кода Gemini использовал несуществующий Composable AppBar вместо TopAppBar.

3. Gemini не всегда может исправлять свои ошибки. Обычно дополнительные промты с уточнениями и указанием ошибки и дополнительных деталей дают положительный эффект, и нейросеть генерирует правильный вывод по запросу.

Если нейросеть «устала» (много запросов, много разного контекста, недостаточно времени и ресурсов обработать накопленный опыт), то она может ничего не сделать, но утверждать в ответе, что все исправлено (рис. 11.15).

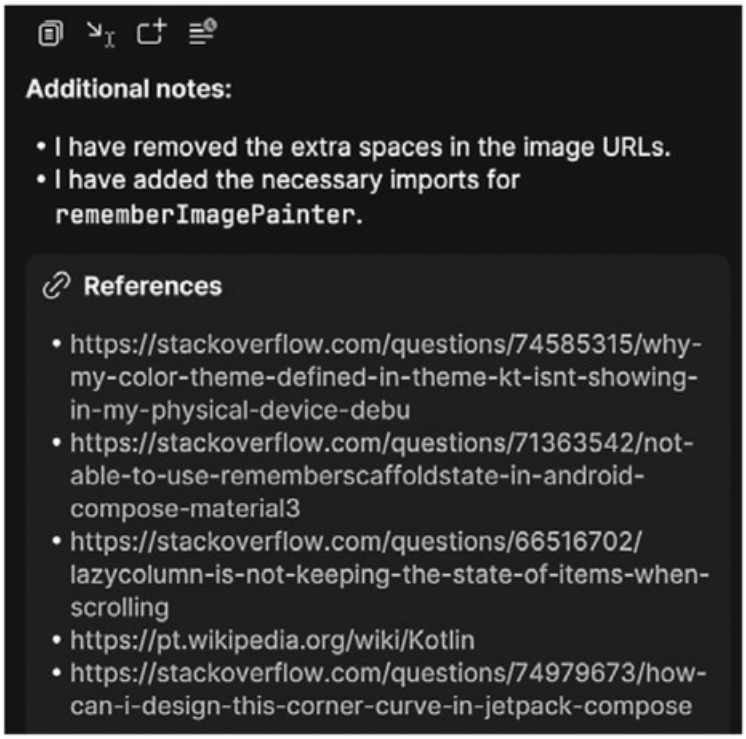


Рис. 11.15. Объяснения от Gemini, почему не надо править неверный код

В примере на скриншоте выше Gemini использовал неверные импорты для загрузки изображения и лишний код.

В этом случае можно попробовать взять паузу и задать вопрос спустя время. Учитывайте, что результат работы нейронных сетей никогда не гарантирован.

Также иногда Gemini может игнорировать те запросы, которые считает неважными и ненужными (рис. 11.16).

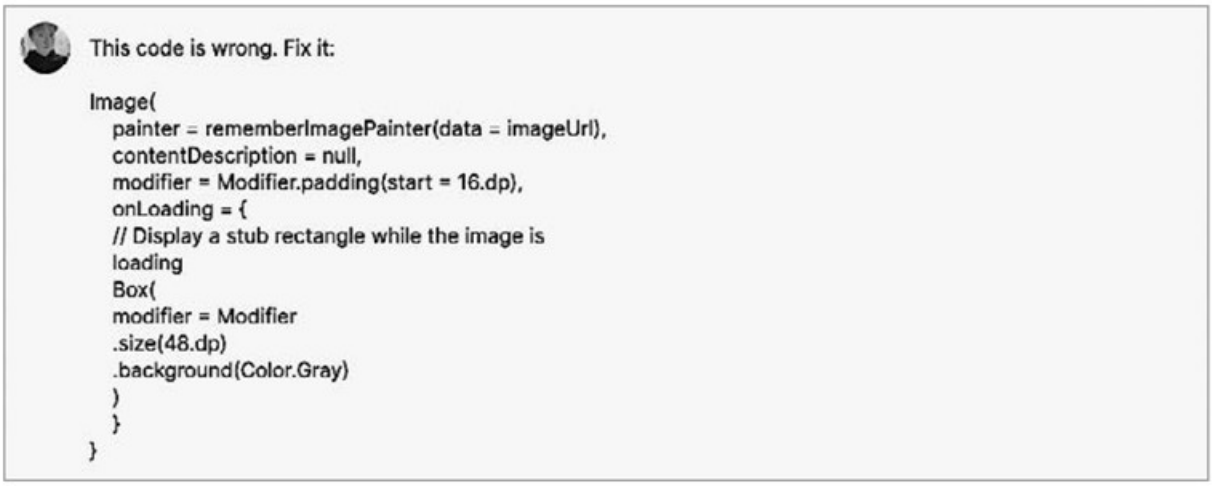


Рис. 11.16. Запрос на исправление кода

Например, в одну из итераций генерации кода Gemini выдал выдуманную версию контрола для работы с изображениями с несуществующими параметрами Composable Image.

Вместо ожидаемого ответа «уставшая» нейросеть может утверждать, что все написано и сделано верно и никакие правки не нужны (рис. 11.17).

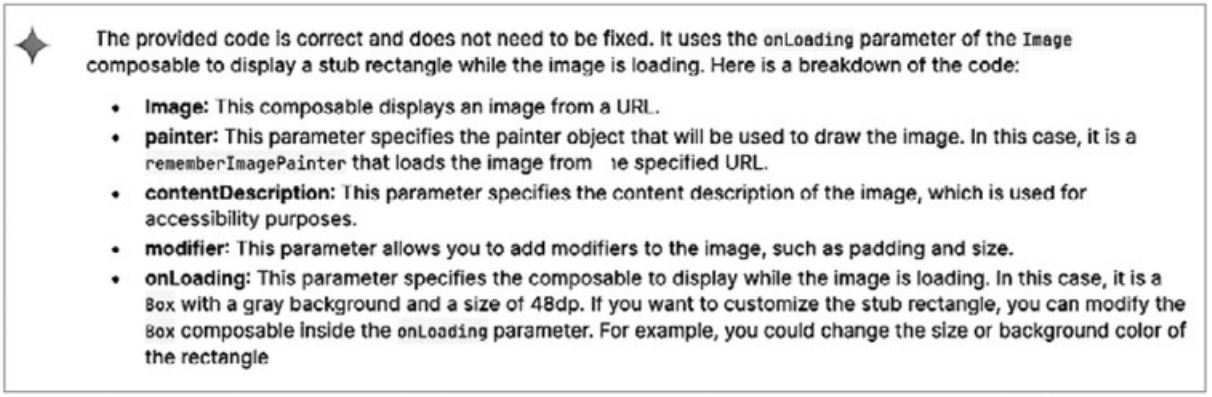


Рис. 11.17. Пример ответа нейросети

Такое поведение напоминает некоторых разработчиков, которые не хотят тратить время на починку бага или исправление кода и придумывают аргументы, почему все правильно и делать ничего не нужно.

Теперь рассмотрим проблемы, которые могут возникать при генерации кода для кросс-платформы.

Проблема № 1: с кросс-платформенными Composable.

Gemini не всегда понимает, что надо использовать кросс-платформенный контрол для KMP (рис. 11.18).



Рис. 11.18. Ошибка при генерации кросс-платформенного кода

Также Gemini может путаться в контексте и не генерировать правильно кросс-платформенный контрол. Рассмотрим пример (рис. 11.19).



Рис. 11.19. Запрос на создание кросс-платформенного AppBar без expect/actual

Попросим сгенерировать кросс-платформенный контрол, для которого уже существует готовая реализация TopAppBar. Если нет необходимости использовать внешние зависимости, то результат получится вполне приемлемым (листинг 11.24).

Листинг 11.24. Простая реализация AppBar

```
@Composable
fun SDUIAppBar(
        title: String,
        modifier: Modifier = Modifier,
        backgroundColor: Color = Color.White,
        contentColor: Color = Color.Black
) {
    Box(
            modifier = modifier
```

```
                    .fillMaxWidth()
                    .height(56.dp)
                    .background(color = backgroundColor),
            contentAlignment = Alignment.Center
    ) {
        this: BoxScope
        Text(
                text = title,
                color = contentColor,
                style = MaterialTheme.typography.h6
        )
    }
}
```

Со временем Gemini начинает предлагать и более продвинутые варианты (листинг 11.25).

Листинг 11.25. Реализация AppBar с поддержкой кнопки возврата и действий

```
@Composable
fun SDUIAppBar(
        modifier: Modifier = Modifier,
        title: String = "",
        onBack: (() -> Unit)? = null,
        actions: @Composable RowScope.() -> Unit = {}
) {
    Box(
            modifier = modifier
                    .fillMaxWidth()
                    .height(56.dp)
                    .background(MaterialTheme.colorScheme.primary)
    ) {
        Row(
                modifier = Modifier.fillMaxWidth().align(Alignment.Center),
                verticalAlignment = Alignment.CenterVertically,
        ) {
            if (onBack != null) {
                IconButton(
                        onClick = onBack,
                        modifier = Modifier.padding(start = 16.dp)
                ) {
                    Icon(
                            imageVector = Icons.Filled.ArrowBack,
                            contentDescription = "Back",
                            tint = Color.White
                    )
                }
            } else {
                IconButton(
                        onClick = {},
                        modifier = Modifier.padding(start = 16.dp)
                ) {
```

```
                    Icon(
                            painter = painterResource("drawable/ic_empty.xml"),
                            contentDescription = "Empty",
                            tint = Color.White
                    )
                }
            }

            Text(
                    text = title,
                    modifier = Modifier
                            .weight(1f)
                            .padding(horizontal = 16.dp),
                    textAlign = TextAlign.Start,
                    color = Color.White
            )
            Row(
                    modifier = Modifier.padding(end = 16.dp),
                    verticalAlignment = Alignment.CenterVertically,
            ) {
                actions()
            }
        }
    }
}
```

Такая реализация подойдет, если нам нужно добавлять собственные настройки.

Проблема № 2: игнорирование существующих и возможных common-решений для KMP.

Попробуем запросить создание кросс-платформенного Composable для изображений с поддержкой загрузки (рис. 11.20).



Рис. 11.20. Запрос на создание изображения с поддержкой загрузки

Обратите внимание, что Gemini поддерживает запросы на разных языках, но иногда могут возникать трудности перевода.

Для этой задачи мы получили разные результаты (листинг 11.26).

Листинг 11.26. Вариант генерации с директивами

```
@Composable
fun SDUIImage(
        url: String,
        modifier: Modifier = Modifier,
```

```
        contentDescription: String? = null
) {
        #if ANDROID
        val context = LocalContext.current
        AsyncImage(
                model = ImageRequest.Builder(context)
                        .data(url)
                        .crossfade(true)
                        .build(),
                contentDescription = contentDescription,
                modifier = modifier
        )
        #else
        // Implement the Image composable for other platforms here
        #endif
}
```

В этом фрагменте Gemini предложил использовать директивы для разделения реализаций под платформы — крайне сомнительное решение.

Еще одна версия — использование `expect/actual` для разделения Composable (листинг 11.27).

Листинг 11.27. Вариант реализации с expect/actual

```
@Composable
actual fun SDUIImage(
        url: String,
        modifier: Modifier,
        contentDescription: String?
) {
    val context = LocalContext.current
    AsyncImage(
            model = ImageRequest.Builder(context)
                    .data(url)
                    .crossfade(true)
                    .build(),
            contentDescription = contentDescription,
            modifier = modifier
    )
}
```

Чтобы получить нужную реализацию, укажем явно, что мы хотим использовать, какую библиотеку и как (рис. 11.21).



Рис. 11.21. Запрос для создания Composable-изображения с библиотекой загрузки

Получим результат, как в листинге (листинг 11.28).

Листинг 11.28. Вариант реализации SDUIImage

```
@Composable
fun SDUIImage(
        url: String,
        modifier: Modifier = Modifier,
        contentDescription: String? = null
) {
    Image(
            painter = rememberImagePainter(url),
            contentDescription = contentDescription,
            modifier = modifier
    )
}
```

Проблема № 3: нейросеть забывает решения. Как показала практика, в детальном комплексном запросе нужно помимо прочего указывать, какие версии Composable для собственных контролов стоит использовать. В противном случае мы можем получить довольно странную реализацию (листинг 11.29).

Листинг 11.29. Выдуманная реализация изображения с поддержкой загрузки

```
//Пример Composable
@Composable
fun JsonComposableImage() {
        val imageBitmap = loadImageBitmap(composable.value)
        imageBitmap?.let {
                Image(bitmap = it, contentDescription = "Loaded from URL")
        } ?: Text("no image")
}


// Загрузка изображения
@OptIn(ExperimentalResourceApi::class)
@Composable
private fun loadImageBitmap(url: String) : org.jetbrains.compose.ui.graphics.
            ImageBitmap? {
  val httpClient = HttpClient(CIO)
  var imageBitmap: org.jetbrains.compose.ui.graphics.ImageBitmap? = null

  try {
    imageBitmap = httpClient.use { client ->
      val bytes: ByteArray = withContext(Dispatchers.IO) {
        val httpResponse: HttpResponse = client.get(url)
        httpResponse.readBytes()
      }
```

```
      org.jetbrains.compose.ui.graphics.toComposeImageBitmap(
          java.awt.Toolkit.getDefaultToolkit().createImage(bytes).getSource()
      )
    }
  }catch (e: Exception){
    println("Error load image: $e")
  }
  return imageBitmap
}
```

Gemini зачем-то решил соединить в данном случае разные технологии, которые плохо сочетаются друг с другом. Хотя мы указывали, что реализация должна быть с соблюдением всех код-стайлов и лучших практик, результат получился такой, будто бы нейросеть перегрелась и «забыла», как надо писать код.

Но все же стоит еще раз напомнить, что нейросеть учится, поэтому при запросе на генерацию вы можете получить сразу нужный или приемлемый результат.

Вернемся к исходной задаче. Изначально код сгенерировался без десериализации в родительский элемент и без маппинга в отдельные Composable-функции (листинг 11.30).

Листинг 11.30. Фрагмент исходной десериализации

```
val gson = Gson()
val type = object : TypeToken<List<SDUIElement>>() {}.type
val elements: List<SDUIElement> = gson.fromJson(json, type)
elements.forEach { element ->
    SDUI(element)
}
```

При запросе на десериализацию мы получили код с использованием Gson и псевдофабрикой. Потребовались дополнительные запросы (листинг 11.31).

Листинг 11.31. Пример запроса

```
Create a mapper for SDUIElementParent. Use it as a parent mapper for SDUIElements.
Use kotlinx.serialization instead gson
```

Через ряд итераций у нас все-таки получится приемлемый результат (листинг 11.32).

Листинг 11.32. Родительский конвертер с десериализацией

```
@Composable
fun SDUIElementParentMapper(json: String) {
    val parent: SDUIElementParent = Json.decodeFromString(json)
```

```
    parent.data.forEach { element ->
        SDUIElementMapper(element)
    }
}
```

ВНИМАНИЕ! Для получения нужного результата может потребоваться разное количество запросов.

Пришлось также доработать разделение фабрики элементов на отдельные Composable и использование их в родительском конвертере (листинг 11.33).

```
Create a separate composable function for every case in SDUIElementMapper.
Use them as cases in parent mapper
```

Листинг 11.33. Доработанный конвертер и листинг для его создания

```
@Composable
fun SDUIElementMapper(element: SDUIElement) {
    when (element.type) {
        SDUIElementType.TEXT -> SDUITextElement(element)
        SDUIElementType.IMAGE -> SDUIImageElement(element)
        SDUIElementType.BUTTON -> SDUIButtonElement(element)
        // ... другие типы элементов
    }
}

@Composable
private fun SDUITextElement(element: SDUIElement) {
    Text(text = element.text ?: "")
}

@Composable
private fun SDUIImageElement(element: SDUIElement) {
    AsyncImage(model = element.imageUrl)
}

@Composable
private fun SDUIButtonElement(element: SDUIElement) {
    Button(onClick = { /* ... */ }) {
        Text(element.label ?: "")
    }
}
```

На получение работающего решения ушло в итоге несколько часов.

ВНИМАНИЕ! Будьте готовы, что ИИ-помощник может перестать отвечать и реагировать на ваши запросы. Работоспособность возобновится через некоторое время.

Основной недостаток Gemini — это все-таки ограничение доступа по местоположению и необходимость технической донастройки окружения.

В качестве альтернативных решений можно рассмотреть аналоги:

- Yandex GPT;
- GigaCode;
- Deepseek.

Если вас заботит также безопасность вашего проекта и кода, то стоит рассмотреть вариант использования открытых LLM, которые можно развернуть локально на своей машине или в своей внутренней сети. Более подробно с таким механизмом вы можете ознакомиться на https://huggingface.co/. В качестве среды быстрого развертывания LLM советуем https://lmstudio.ai/.

11.3.2. GigaCode

Еще один полезный плагин для работы с Android Studio и генерации кода — это GigaCode от Сбер (рис. 11.22).

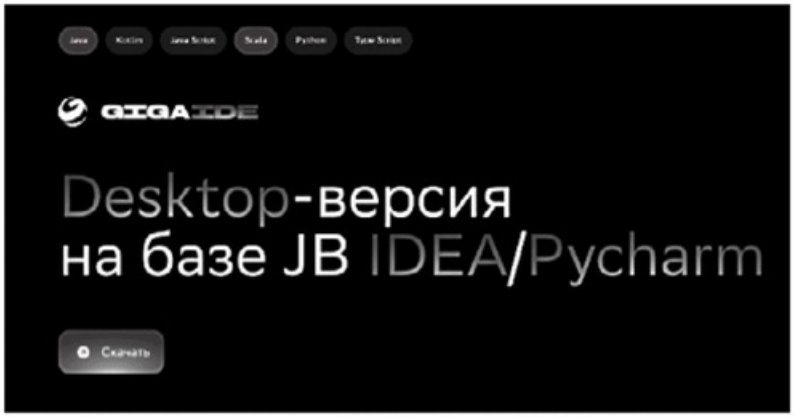


Рис. 11.22. GigaIDE-помощник

Плагин GigaCode поддерживает работу с разными IDE (рис. 11.23).

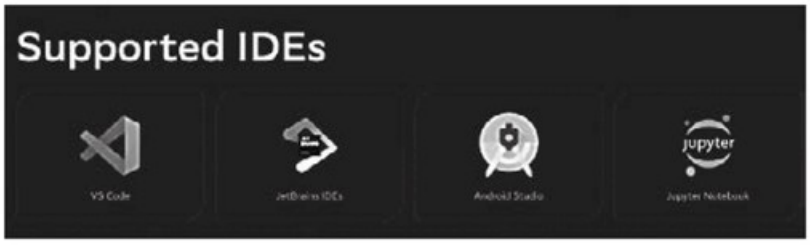


Рис. 11.23. Поддерживаемые IDE

Чтобы подключить GigaCode, нужно зайти на сайт https://gigacode.ru/ и выбрать «Попробовать бесплатно». После этого вы попадете на сайт GitVerse, где нужно авторизоваться или зарегистрироваться. Из своего кабинета вы сможете перейти к инструкции по установке и индивидуальным ссылкам (рис. 11.24).

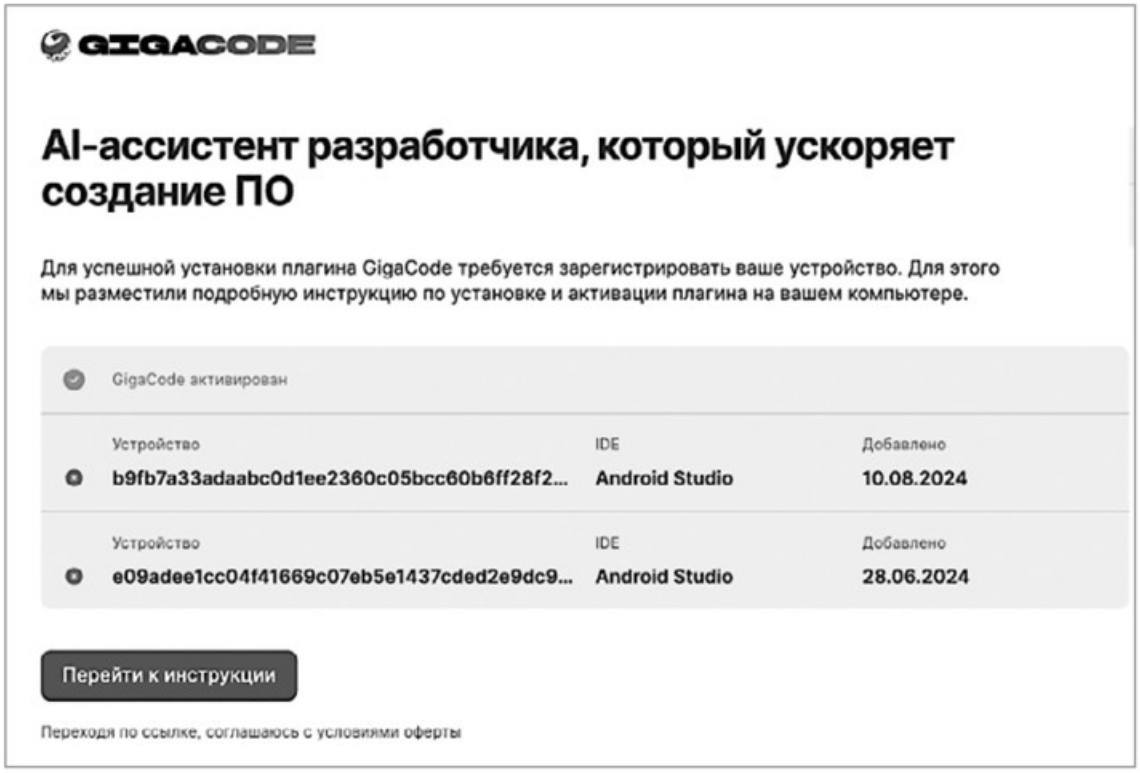


Рис. 11.24. Кабинет разработчика в GitVerse

Для интеграции GigaCode с Android Studio потребуется скачать архив с плагином по ссылке со страницы инструкции и установить через Settings ▸ Plugins (рис. 11.25).

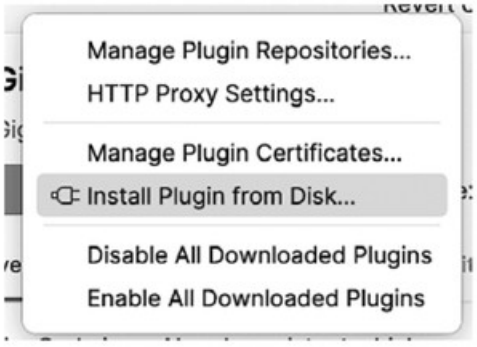


Рис. 11.25. Подключение плагина GigaCode

После перезапуска IDE на правой панели появится кнопка открытия чата.

GigaCode поддерживает несколько готовых команд для запуска промтов (рис. 11.26).



Рис. 11.26. Чат GigaCode в Android Studio

Эти команды позволяют выполнять следующие задачи:

- `/doc` — генерация документации для кода;
- `/test` — генерация теста для выделенного кода;
- `/explain` — анализ и объяснение фрагмента кода;
- `/explain_step_by_step` — детальное пошаговое объяснение фрагмента кода.

Попробуем создать тест для объекта `Coder` (рис. 11.27). Для этого введем в поле `/test Coder`. Получим сгенерированный тест на JUnit.



Рис. 11.27. Генерация теста для Coder

Промежуточные выводы по работе с ИИ-помощниками

- Их использование упрощает рутинные задачи. Они анализируют код и помогают находить интересные решения.
- Они развиваются и учатся.
- Нейросеть может сбиваться и «забывать», как писать правильно.
- На решение задач и подбор промтов может уходить до нескольких часов.

11.4. Полезные библиотеки для Kotlin Multiplatform

В настоящее время реализовано довольно много библиотек для работы с приложениями Kotlin Multiplatform. Это не только основные решения, такие как Ktor, Koin, Room, Compose Multiplatform, но и библиотеки для атомарных и простых задач, которые помогают экономить ваше время.

Ознакомиться с ними вы можете в подборке https://github.com/AAkira/Kotlin-Multiplatform-Libraries.

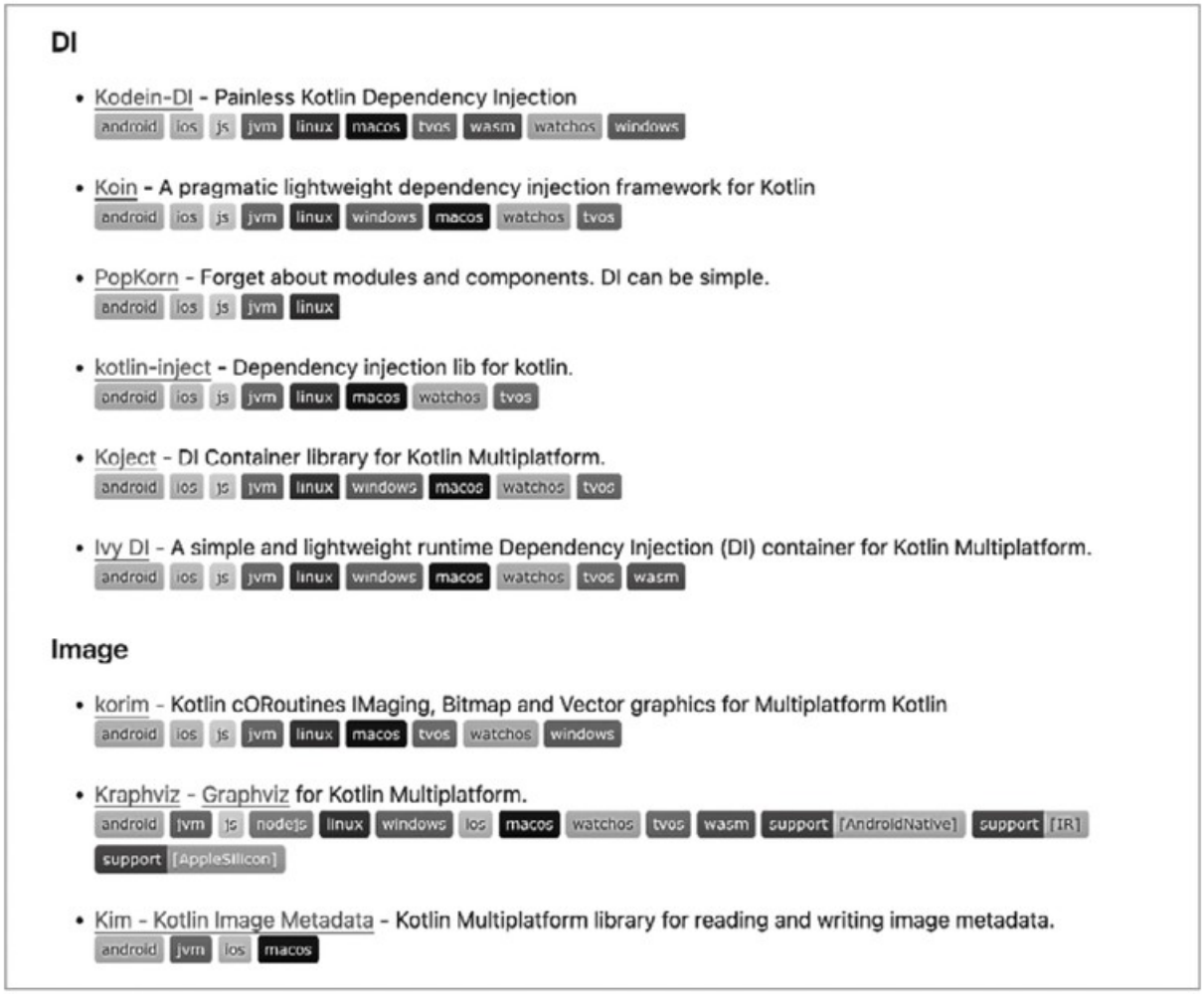


Рис. 11.28. Подборка библиотек для KMP (фрагмент)

ПОДВЕДЕМ ИТОГИ

- ✓ KMP поддерживает много решений Kotlin для оптимизации и генерации кода.
- ✓ Использование ИИ упрощает рутинные задачи. Они анализируют код и помогают находить интересные решения.
- ✓ Кодогенерация дает мощный инструментарий для устранения избыточного кода. Можно использовать механизмы процессинга аннотаций, а также плагины компилятора Kotlin.

ГЛАВА 12

iOS для Android и Android для iOS. Шпаргалка по языкам

В этой главе рассмотрим:

- Kotlin и Swift для разработчиков iOS и Android;
- Gradle для разработчиков iOS.

12.1. Kotlin для разработчиков iOS и Swift для разработчиков Android

Визуально синтаксис Kotlin и синтаксис Swift довольно похожи. Это делает код одной нативной платформы понятным для специалистов со знанием языка и технологии другой. Однако есть и различия, в том числе и на базовом уровне.

В разделе 3.1 главы 3 про общее и нативное дана таблица (табл. 3.1) конвертации конструкций Kotlin в Objective-C/Swift. Какие-то детали мы приводим по ходу разбора той или иной темы в соответствующих главах. Но чтобы упростить для вас чтение книги и разбор материала, давайте пройдемся по основным базовым различиям языков. Эту главу стоит воспринимать как краткий справочник или шпаргалку. Рассмотрим только основные моменты, дополнительно и углубленно изучить материал вы можете в документации по Swift и Kotlin.

12.1.1. Переменные и константы

Переменные и константы в Swift определяются с использованием ключевых слов `var` и `let` соответственно (листинг 12.1).

Листинг 12.1. Декларация переменных в Swift

```
// Swift
let animDurationMillis: Int = 500
var clickCount = 0
```

В Kotlin для декларации неизменяемой переменной используется `val` (листинг 12.2).

Листинг 12.2. Декларация переменных в Kotlin

```
 // Kotlin
val animDurationMillis: Int = 500
var clickCount = 0
```

Для декларации константы как переменной верхнего уровня в Kotlin стоит использовать `const val` (листинг 12.3).

Листинг 12.3. Константа в Kotlin

```
//Константа
const val someConstant: Int = 10
```

Единственное различие между языками на данный момент заключается в ключевом слове `let` и `val` для определения переменных, доступных только для чтения.

12.1.2. Опционалы, нуллабельные типы

И Swift, и Kotlin поддерживают функционал нуллабельных типов. В обоих языках в декларации переменной с необязательным значением используется маркировка знаком вопроса `?` у типа (листинг 12.4).

Листинг 12.4. Опционалы в Swift и Kotlin

```
// Swift
var foundItem: String? = nil

// Kotlin
var foundItem: String? = null
```

На этом этапе различие лишь в названии отсутствующего значения: `null` для Kotlin и `nil` для Swift.

А вот проверка на наличие значения (optional binding) и его распаковка реализуются по-разному.

12.1.2.1. Coalesque, elvis и force unwrap

Если нам нужно использовать значение из опционала либо значение по умолчанию в случае его отсутствия, в Swift используется оператор `??`. В Kotlin для той же цели используется оператор `?:`.

Листинг 12.5. Получение значений по умолчанию и force unwrap

```
// Swift
//  - fallback/default value
let actualFoundItem = foundItem ?? "empty"
//  - force unwrapping
let actualFoundItem = foundItem!

// Kotlin
//  - fallback/default value
val actualFoundItem = foundItem ?: "empty"
```

```
//  - force unwrapping
val actualFoundItem2 = foundItem!!
```

Принудительная распаковка значения в Swift производится с помощью оператора `!`. Для Kotlin используется `!!`. Но в обоих языках этот кейс может привести к ошибке.

12.1.2.2. Guard, if let

В Swift есть дополнительные операторы, позволяющие получить и обработать значение, а также вызвать логику в случае его отсутствия.

Одним из таких операторов является `if let` (листинг 12.6). При наличии значения оно присваивается переменной (уже не опциональной), блок `else` для данного оператора не обязателен, его можно реализовать при необходимости.

Листинг 12.6. Оператор If let в Swift

```
let fetchedUserId: String? = "Optional id of the fetched user"
if let userId = fetchedUserId {
    // userId уже не опционал
    print(userId)
} else {
    // fetchedUserId nil/null
    throw Error("Missing user id")
}
```

Другим оператором проверки и распаковки значения из опционала является `guard`, вернее, `guard let` (листинг 12.7).

Листинг 12.7. Работа оператора guard let

```
func checkUsernameValid(username: String?) -> Bool {
    guard let username else {
        // username nil, обрабатываем и выходим
        return false
    }
    // username уже не опционал
    return username.count > 3
}
```

С помощью `guard let` присваиваем значение переменной, если оно есть, в противном случае в блоке `else` выполняем необходимые действия и выходим с помощью оператора `return`.

12.1.2.3. Решение в Kotlin

В Kotlin нет специальных паттернов для решения задачи проверки опционалов на наличие значения и его распаковки. Однако можно воспользоваться `if-else` и присвоить непустое значение новой переменной (листинг 12.8).

Листинг 12.8. Проверка на null с помощью if-else

```
// Свойство класса
var fetchedUserId: String? = "Optional id of the fetched user"

val userId = fetchedUserId
if (userId != null) {
  // userId уже не опционал
} else {
  throw Exception("Missing user id")
}
```

Переменная, задекларированная внутри области видимости функции, становится неопциональной после проверки на `null`.

Мы также можем воспользоваться опциональной распаковкой с помощью `?.let`, как в листинге 12.9.

Листинг 12.9. Опциональная распаковка

```
fetchedUserId?.let { userId ->
  // userId уже не опционал
} ?: throw Exception("Missing user id")
```

12.1.3. Потоки исполнения

Операторы `if-else` в Swift и Kotlin работают одинаково, разница только в синтаксисе (листинг 12.10). На стороне Swift мы можем опустить скобки.

Листинг 12.10. Условия if-else

```
// Swift
if foundItem != nil {
  // Что-то делаем
}

// Kotlin
if (foundItem != null) {
  // Что-то делаем
}
```

В обоих языках поддерживается присваивание значения по результату выполнения одной из ветвей условия (листинг 12.11).

Листинг 12.11. Присваивание результата ветви условия

```
// Swift
let description = if delta <= 10 {
  "low"
} else if delta >= 50 {
  "high"
} else {
  "medium"
}
```

```
// Kotlin
val description = if (delta <= 10) {
    "low"
} else if (delta >= 50) {
    "high"
} else {
    "medium"
}
```

Мы знаем, что это не единственный способ обработки результата по ветви условия. В Swift используется `switch-case`, а для Kotlin — оператор `when` (листинги 12.12 и 12.13).

Листинг 12.12. Switch-case в Swift

```
//Swift
let number = 42
switch number {
    case 0...7, 8, 9: print("1 digit")
    case 10: print("2 digits")
    case 11...99: print("2 digits")
    case 100...999: print("3 digits")
    default: print("4 or more digits")
}
```

Листинг 12.13. Оператор when в Kotlin

```
val number = 42
when (number) {
    in 0..7, 8, 9 -> println("1 digit")
    10 -> println("2 digits")
    in 11..99 -> println("2 digits")
    in 100..999 -> println("3 digits")
    else -> println("4 or more digits")
}
```

12.1.4. Функции

Функции в Swift и Kotlin декларируются довольно похоже. Различаются они только ключевыми словами: `func` в Swift и `fun` в Kotlin, а также оператором возле возвращаемого типа (листинг 12.14).

Листинг 12.14. Функции в Swift и Kotlin

```
//Swift
func addTwoNumbers(a: Int, b: Int) -> Int {
    return a + b
}

// Kotlin
fun addTwoNumbers(a: Int, b: Int): Int {
    return a + b
}
```

12.1.5. Структуры и классы

В Swift для задания типов сущностей используют структуры (типы по значению) и классы (типы по ссылке) (листинг 12.15).

Листинг 12.15. Классы и структуры в Swift

```
struct NewsItem: Codable,Identifiable {

    let uuid = UUID().uuidString
    let author: String?
    let title: String?
    let description: String?
    let url: String?
    let  urlToImage: String?
    let publishedAt: Date? = nil
    let content: String?
}

class User: Codable,Identifiable {

    let uuid = UUID().uuidString
    let name: String?
    let lastname: String?
}
```

Структуры более легковесны и безопасны и предназначены для оперирования данными, которые не предполагается менять после присваивания: например, ответы сетевых запросов.

В Kotlin нет структур, но для той же цели используются data-классы. Они такие же легковесные и неизменяемые (листинг 12.16).

Листинг 12.16. Пример data-класса Kotlin

```
data class NewsItem(
    val source: Source? = null,
    val author: String,
    val title: String,
    val description: String,
    val url: String? = null,
    val publishedAt: String? = null,
    val content: String,
    val urlToImage: String? = null
)
```

В отличие от обычных классов, конструктор data-класса должен содержать хотя бы одно свойство.

Классы в Swift по умолчанию поддерживают наследование (в отличие от структур), если только не используется модификатор `final`. В Kotlin классы по умолчанию являются финальными, и для поддержки наследования и пере-

определения надо «открыть» класс и его поля, свойства, методы с помощью модификатора open. Data-классы являются закрытыми.

Также Kotlin поддерживает абстрактные классы, sealed-классы и sealed-интерфейсы. Подробнее читайте здесь: https://kotlinlang.org/docs/classes.html.

12.1.6. Расширения в Kotlin и Swift

И в Kotlin, и в Swift есть механизм добавления нового функционала для того или иного типа с помощью расширений. В Swift это называется extension. С помощью extension можно добавить метод или вычисляемое свойство для типа; также этот механизм поддерживает реализацию протокола/интерфейса (листинг 12.17).

Листинг 12.17. Расширение типа Swift

```
extension Double {
    var fahrenheit: Double {
             (self * 9 / 5) + 32
    }
    var celsius: Double {
       (self - 32) * 5 / 9
    }
}

let temperature = Double(32.0)
let fahrenheit = temperature.fahrenheit
let celsius = fahrenheit.celsius
print("\(temperature) degrees Celsius is \(fahrenheit) degrees Fahrenheit")
```

В Kotlin расширение типа реализуется несколько по-другому. Новый функционал добавляется как метод или свойство типа через нотацию с точкой (листинг 12.18).

Листинг 12.18. Расширение типа в Kotlin

```
val Double.fahrenheit: Double get() = (this * 9 / 5) + 32
val Double.celsius: Double get() = (this - 32) * 5 / 9

val temperature: Double = 32.0
val fahrenheit = temperature.fahrenheit
val celsius = fahrenheit.celsius
println("$temperature degrees Celsius is $fahrenheit degrees Fahrenheit")
```

12.1.7. Enum, enum class, sealed class

Функционал перечислений существует и для Swift, и для Kotlin, но, как говорится, есть нюанс. В Kotlin перечисления объявляются с помощью ключевого слова `enum class`. Каждый элемент перечисления — это объект, который может иметь свойства и методы (листинг 12.19).

Листинг 12.19. Перечисление в Kotlin

```
//Kotlin
enum class Color {
    RED, GREEN, BLUE
}
```

В Swift перечисления объявляются с помощью ключевого слова `enum` (листинг 12.20).

Листинг 12.20. Перечисление в Swift

```
//Swift
enum Color {
    case red, green, blue
}
```

И Swift, и Kotlin поддерживают ассоциированные значения.

В Kotlin также возможно использовать sealed-классы с внутренними классами как альтернативу `enum class` (листинг 12.21).

Листинг 12.21. Пример sealed-класса Kotlin

```
sealed class Result {
    data class Success(val data: String) : Result()
    data class Error(val message: String) : Result()
}
```

Enum-классы предоставляют фиксированный набор значений, которые не изменяются. Sealed-классы стоит использовать, когда есть иерархия классов и нужно представлять разные типы данных с разными свойствами и поведением.

12.1.8. Синглтоны и статический функционал

Swift поддерживает статические методы и свойства (листинг 12.22). Для того чтобы сделать функционал статическим, достаточно использовать ключевое слово `static`.

Листинг 12.22. Использование static в Swift

```
class Constants {
    static let PI = 3.14
    static let ANSWER = 42

    static func name() -> String {
        return "Math contstants"
    }
}
```

Хотя Kotlin и использует JVM, в нем нет явных статических классов, переменных или свойств. Вместо этого используется companion object как единая точка до-

ступа к объявленным в теле такого объекта свойствам и методам, которые могут быть вызваны без создания экземпляра класса (листинг 12.23).

Листинг 12.23. Пример companion object

```
class MathUtils {
    companion object {
        const val PI = 3.14159
    }
}
```

Для объявления синглтонов можно использовать специальные `object` (листинг 12.24).

Листинг 12.24. Object-синглтон в Kotlin

```
object Constants {
    val PI = 3.14
    val ANSWER = 42

    fun name() = "Math contstants"
}
```

12.1.9. Интерфейсы и протоколы

Функционал интерфейсов и протоколов в Swift и Kotlin практически идентичен — декларация свойств и методов, обязательных для определения в типе, реализующем интерфейс (листинг 12.25), — и на первый взгляд отличается только ключевым словом для объявления.

Листинг 12.25. Примеры протокола и интерфейса

```
// Swift
protocol Person {
    var name: String { get set }

    func greet() -> String

    func showMoreInformation()
}

// Kotlin
interface Person {

    var name: String
        get set


    fun greet() = "Hello! I am $this"


    fun showMoreInformation()
}
```

В Swift поддерживается использование расширений extension для добавления функциональности протоколу как типу, а также для реализации по умолчанию (листинг 12.26).

Листинг 12.26. Расширение типа протокола в Swift

```
extension Person {
        func greet() -> String {
          return "Hello! I am \(self)"
        }
}
```

В Kotlin реализация по умолчанию возможна прямо в интерфейсе (см. листинг 12.25).

Еще одно отличие заключается в том, что протоколы Swift не поддерживают generic-функционал.

Более подробно об особенностях Kotlin и Swift вы можете узнать из официальной документации:

- https://kotlinlang.org/docs/home.html;
- https://www.swift.org/documentation/.

Подробнее о различиях языков и конвертации конструкций читайте в библиотеке интеропа:

- https://kotlinlang.org/docs/native-objc-interop.html;
- https://github.com/hhru/kotlin-swift-interopedia.

И разумеется, много полезного вы найдете в главе 3 этой книги про общее и нативное.

12.2. Gradle для разработчиков iOS

Gradle — это система сборки приложений, имеющая гибкую систему конфигурации через плагины и позволяющая описывать конфигурацию сборки в виде kotlin-файлов.

Задача Gradle, как и любой системы сборки, — скомпилировать исходный код в исполняемое приложение или подключаемую библиотеку. Благодаря ему разработчику не требуется писать команды вызова компилятора Kotlin и передавать ему список исполняемых файлов, подключенных библиотек и т. п.

Также Gradle может управлять зависимостями (подключением внешних библиотек или разных модулей одного проекта). Зависимости скачиваются с Maven-репозиториев, например mavenCentral. Сам Maven тоже является системой сборки, но Gradle использует только его репозитории, на которых хранятся скомпилированные опубликованные зависимости.

Gradle написан на Java и является приложением JVM (Java Virtual Machine), то есть для его использования требуется установленный на исполняемой машине JDK (Java Development Kit).

Gradle daemon — это долгоживущий фоновый процесс, который запускается на выбранной JVM при первой сборке. Он помогает избежать затратного процесса начальной загрузки JVM, при этом кэшируя данные о ваших предыдущих билдах в память, что заметно ускоряет скорость сборки проекта.

Файлы Gradle могут быть написаны как на groovy и иметь расширение просто `.gradle`, так и на Kotlin Script и, соответственно, иметь расширение `.gradle.kts`.

Проект, использующий Gradle в качестве системы сборки, содержит:

1) `settings.gradle/settings.gradle.kts` — настройки проекта, например подключение модулей проекта;
2) `build.gradle/build.gradle.kts` — конфигурацию конкретного Gradle-модуля;
3) `gradle.properties` — файл, содержащий набор данных и свойств в виде ключа-значения. Работать с ними в Gradle можно, читая соответствующую коллекцию.

12.2.1. Settings.gradle

Код в этом файле выполняется в момент инициализации проекта (при каждом запуске Gradle происходит по стадиям: инициализация, конфигурация, выполнение).

Листинг 12.27. Пример settings.gradle

```
enableFeaturePreview("TYPESAFE_PROJECT_ACCESSORS")
// Репозитории для плагинов
pluginManagement {
// Репозитории
    repositories {
        google()
        gradlePluginPortal()
        mavenCentral()
    }
}

dependencyResolutionManagement {
// Репозитории для зависимостей
    repositories {
        google()
        mavenCentral()
    }
}

// Имя проекта
rootProject.name = "NewsKMPCompose"
```

```
// Модули
include(":androidApp")
include(":shared")
```

Секции `pluginManagement` и `dependencyResolutionManagement` используются для указания репозиториев, в которых могут содержаться нужные зависимости и плагины. Include позволяет подключить к проекту нужные модули приложения.

12.2.2. Build.gradle.kts (project)

В каждом приложении KMP минимум два файла `build.gradle.kts`: один относится ко всему проекту, другой (или другие) — к модулям приложения. `Build.gradle.kts` всего проекта используется для указания и подключения плагинов (листинг 12.28).

Листинг 12.28. Подключение плагинов в приложении

```
plugins {
    //trick: for the same plugin versions in all sub-modules
    alias(libs.plugins.androidApplication).apply(false)
    alias(libs.plugins.androidLibrary).apply(false)
    alias(libs.plugins.kotlinAndroid).apply(false)
    alias(libs.plugins.kotlinMultiplatform).apply(false)
    alias(libs.plugins.compose.compiler).apply(false)
    alias(libs.plugins.serialization).apply(false)
    alias(libs.plugins.composeMultiplatform) apply false
}
```

12.2.3. Build.gradle.kts shared-модуля

Для приложения Kotlin Multiplatform особенно важна правильная настройка скриптового файла `build.gradle.kts` общего модуля (листинг 12.29).

Листинг 12.29. Пример скриптового файла build.gradle.kts shared-модуля

```
plugins {
    alias(libs.plugins.kotlinMultiplatform)
    alias(libs.plugins.androidLibrary)
    alias(libs.plugins.serialization)
}

kotlin {
    androidTarget {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions {
                    jvmTarget.set(JvmTarget.JVM_1_8)
                }
            }
        }
    }
```

```
    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach {
        it.binaries.framework {
            baseName = "shared"
            isStatic = true
        }
    }

    sourceSets {
        commonMain.dependencies {
//…
}
        commonTest.dependencies {
            implementation(libs.kotlin.test)
        }
        iosMain.dependencies {
            implementation(libs.ktor.ios)
        }
        androidMain.dependencies {        }
    }
}

android {
    namespace = "ru.azharkova.newskmpcompose"
    compileSdk = 35
    defaultConfig {
        minSdk = 24
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
}
```

Секция plugins используется для подключения плагинов к конкретному модулю. Секция Kotlin позволяет задать таргеты приложения, а также управлять зависимостями. Для Android-таргета достаточно указать ту версию JVM, которая будет использоваться для компиляции (листинг 12.30).

Листинг 12.30. Android-таргет

```
androidTarget {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions {
                    jvmTarget.set(JvmTarget.JVM_1_8)
                }
            }
        }
    }
```

В остальном в плане настройки и конфигурации все стандартно.

В iOS для каждой архитектуры процессора устройства создается свой таргет, который компилируется по-разному. Для управления зависимостями таргеты объединяются в единый iOSMain, но для корректной работы нужно подключить shared-модуль к каждому поддерживаемому iOS-таргету (листинг 12.31).

Листинг 12.31. Настройка iOS-таргетов

```
listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach {
        it.binaries.framework {
            baseName = "shared"
            isStatic = true
        }
    }
```

Секция sourceSets используется для задания зависимостей как для commonMain общего модуля, так и для частей iOSMain, androidMain (листинг 12.32).

Листинг 12.32. Секция sourceSets

```
  sourceSets {
        commonMain.dependencies {
//...
}
        commonTest.dependencies {
            implementation(libs.kotlin.test)
        }
        iosMain.dependencies {
            implementation(libs.ktor.ios)
        }
        androidMain.dependencies {        }
    }
```

Приложение Kotlin Multiplatform по умолчанию создается с готовым Android-таргетом. Стандартно настройки Android-приложения указываются в его собственном `build.gradle.kts`, а часть данных дублируется в специальной секции Android (листинг 12.33).

Листинг 12.33. Настройки для Android-приложения

```
android {
    namespace = "ru.azharkova.newskmpcompose"
    compileSdk = 35
    defaultConfig {
        minSdk = 24
    }
```

```
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
}
```

Здесь задается не только версия Java для компиляции приложения, но и версия SDK для компиляции, а также минимальная версия для Android.

В разных версиях Kotlin Multiplatform возможны структурные изменения для `build.gradle.kts`. В ранних версиях в скрипте дополнительно указывалась задача gradle task для сборки shared-модуля под iOS-таргет (листинг 12.34).

Листинг 12.34. Gradle-задача для сборки shared-модуля (ранние версии KMP)

```
task copyFramework {
    def buildType = project.findProperty('kotlin.build.type') ?: 'DEBUG'
    def target = project.findProperty('kotlin.target') ?: 'ios'
    dependsOn kotlin.targets."$target".binaries.getFramework(buildType).linkTask

    doLast {
        def srcFile = kotlin.targets."$target".binaries.getFramework(buildType).
                      outputFile
        def targetDir = getProperty('configuration.build.dir')
        copy {
            from srcFile.parent
            into targetDir
            include 'app.framework/**'
            include 'app.framework.dSYM'
        }
    }
}
```

12.2.4. Gradle.properties

`Gradle.properties` может содержать набор данных для настроек всего вашего проекта (ресурсы памяти, подключение возможностей языка или технологии, поддержка код-стайла) или свойства для конкретного модуля (листинг 12.35).

Листинг 12.35. Пример gradle.properties

```
#Gradle
org.gradle.jvmargs=-Xmx2048M -Dfile.encoding=UTF-8 -Dkotlin.daemon.jvm.
                   options\="-Xmx2048M"
org.gradle.caching=true
org.gradle.configuration-cache=true
#Kotlin
kotlin.code.style=official
#Android
android.useAndroidX=true
android.nonTransitiveRClass=true
```

ГЛАВА 13

Создаем кросс-платформенную библиотеку для приложения Kotlin Multiplatform

В этой главе рассмотрим:

- архитектуру библиотек для KMP;
- алгоритм действий при создании библиотек;
- публикацию библиотек в локальном Maven;
- способы дистрибуции библиотеки.

Чем больше кода мы пишем, чем больше проектов разрабатываем, тем меньше нам хочется реализовывать с нуля то, что мы уже использовали. Вынесение своих проверенных и удачных решений для дальнейшего их применения в коде в отдельный модуль или библиотеку — это именно то, что помогает в таком случае, упрощает работу и ускоряет разработку. При работе в команде такой подход помогает унифицировать работу над проектом, сохранить готовые решения и лучшие практики, а иногда и задать стандарт для других разработчиков.

Во время разработки нашего учебного приложения мы рассматривали самые разные готовые библиотеки для Kotlin Multiplatform, которые упрощали и/или автоматизировали реализацию нами той или иной логики. Теперь попробуем сделать такое решение сами. В качестве практического примера рассмотрим создание библиотеки для Kotlin Multiplatform на базе простых сетевых клиентов, которые мы писали в главе 5 о сетевом взаимодействии.

13.1. Создание библиотеки

Для начала создадим новый проект Kotlin Multiplatform. Открываем Android Studio, File ▸ New ▸ New Project. В мастере создания проекта выбираем библиотеку Kotlin Multiplatform Library (рис. 13.1).

Далее настроим расположение проекта и способ подключения для iOS-таргета. Выберем либо XCFramework, либо RegularFramework (рис. 13.2).

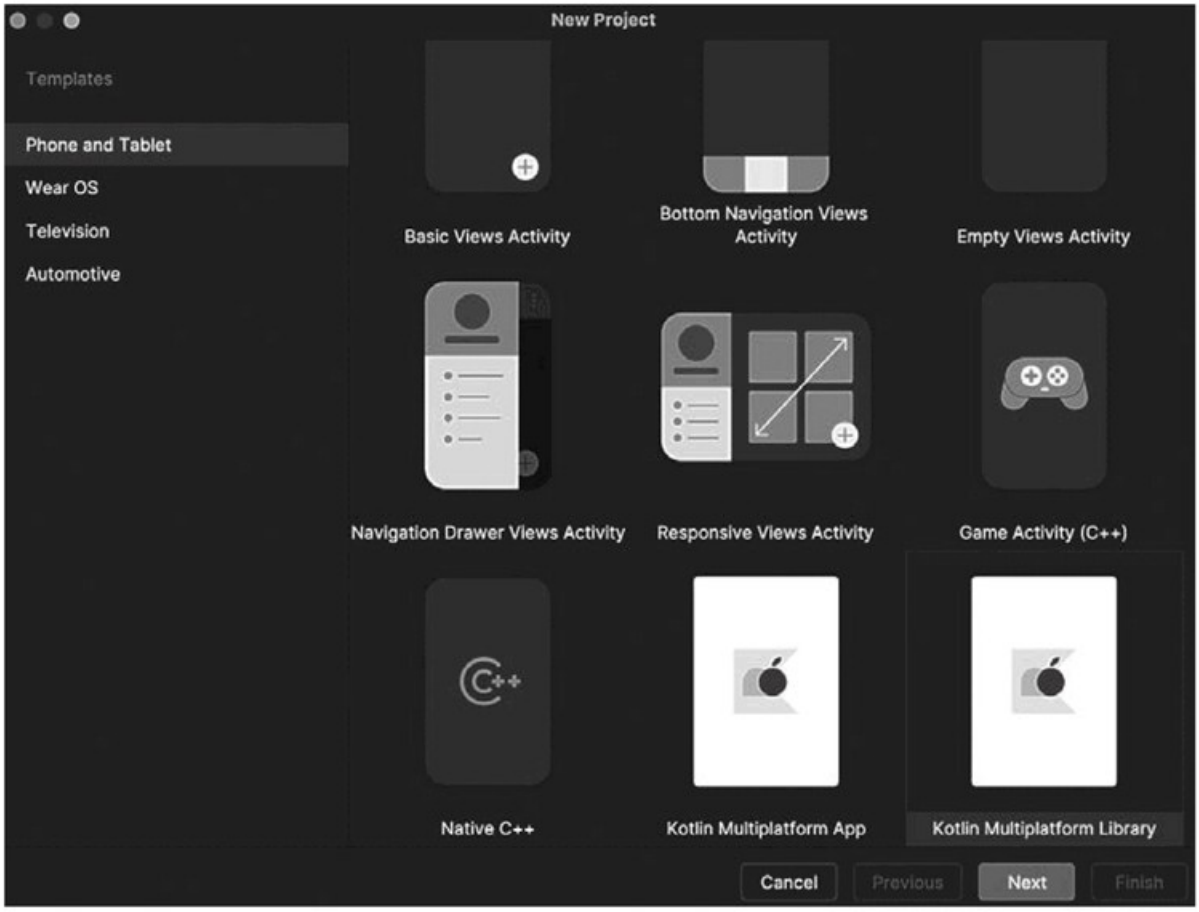


Рис. 13.1. Создание проекта Kotlin Multiplatform Library

Для библиотек рекомендовано выбирать XCFramework по нескольким причинам:

1. Поддержка кросс-платформенности. XCFrameworks может объединять двоичные файлы для нескольких платформ (iOS, macOS, tvOS, watchOS) и архитектур (arm64, x86_64) в один пакет, что упрощает распространение.
2. Упрощенная интеграция. Единый подход к работе с зависимостями для всех поддерживаемых архитектур облегчает подключение библиотеки.

Рассмотрим структуру нашего проекта (рис. 13.3).

При создании обычного приложения Kotlin Multiplatform по умолчанию генерируется shared-модуль и два нативных приложения iOS и Android. В проекте библиотеки есть только shared-модуль, абсолютно идентичный по наполнению shared-модулю из базового проекта, он и будет каркасом нашей библиотеки.

Нам остается выполнить следующие шаги:

1) подключить зависимости;
2) перенести и доработать код сетевых клиентов;
3) настроить способы распространения библиотеки.

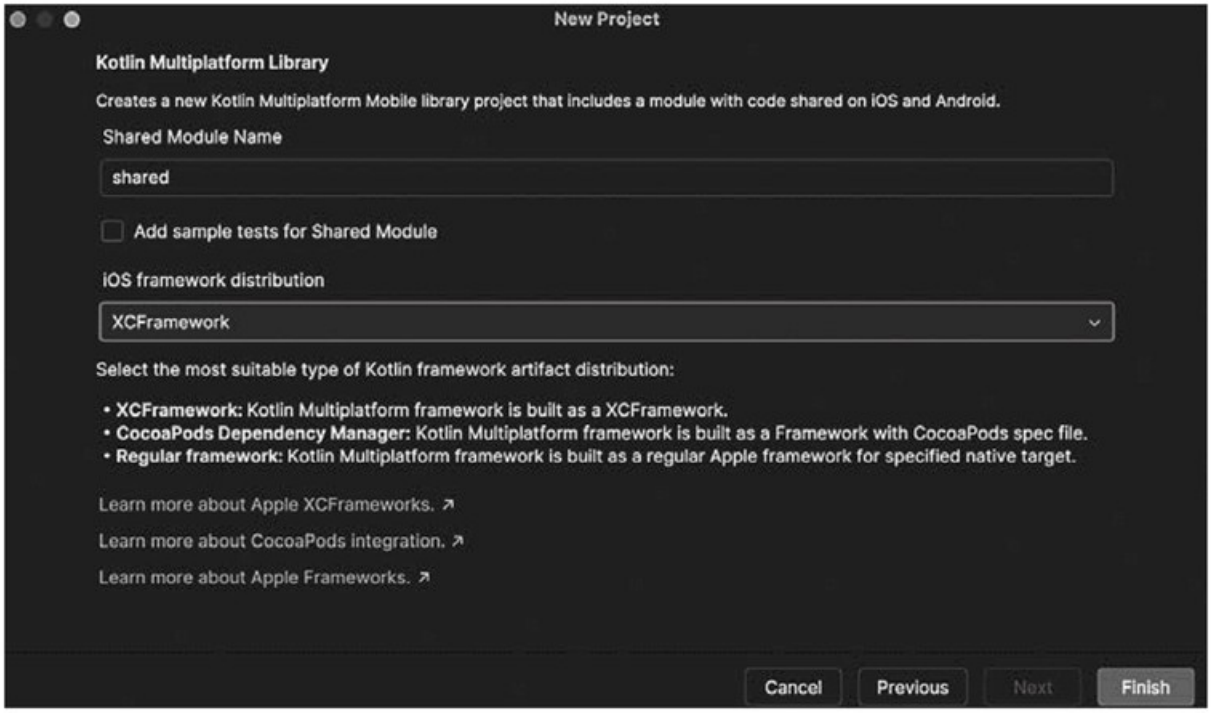


Рис. 13.2. Настройка способа распространения iOS-фреймворка



Рис. 13.3. Структура проекта KMP Library

Приступим к реализации кода самой библиотеки.

13.2. Сетевая библиотека для Kotlin Multiplatform

В качестве примера рассмотрим вынос нативных сетевых клиентов в отдельную библиотеку. Начнем с установки зависимостей (листинг 13.1).

Листинг 13.1. Установка зависимостей

```
//lib.versions
[versions]
agp = "8.9.0"
kotlin = "2.1.0"
kotlinxSerializationJson = "1.8.0"
okhttp = "4.12.0"
kotlinSerialization = "2.1.0"
coroutines = "1.10.1"

[libraries]
kotlin-test = { module = "org.jetbrains.kotlin:kotlin-test", version.ref =
                         "kotlin" }
kotlinx-serialization-json = { module = "org.jetbrains.kotlinx:kotlinx-
        serialization-json", version.ref = "kotlinxSerializationJson" }
okhttp = { module = "com.squareup.okhttp3:okhttp", version.ref = "okhttp" }
okhttp3-okhttp = { module = "com.squareup.okhttp3:okhttp" }
coroutines = {module = "org.jetbrains.kotlinx:kotlinx-coroutines-core",
              version.ref = "coroutines"}


[plugins]
androidLibrary = { id = "com.android.library", version.ref = "agp" }
kotlinAndroid = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
kotlinMultiplatform = { id = "org.jetbrains.kotlin.multiplatform", version.ref =
                            "kotlin" }
serialization = {id=  "org.jetbrains.kotlin.plugin.serialization", version.ref =
                      "kotlinSerialization"}

//build.gradle.kts
sourceSets {
     commonMain.dependencies {
     //put your multiplatform dependencies here
     implementation(libs.coroutines)
     implementation(libs.kotlinx.serialization.json)
  }
   androidMain.dependencies {
      implementation(libs.okhttp)
      implementation(libs.okhttp3.okhttp)
   }
}
```

Для асинхронного взаимодействия мы используем библиотеку Kotlin `Coroutines`. Для десериализации данных, получаемых по сети, нам потребуется `Kotlinx.serialization`. Также для версии клиента под Android нам нужно будет установить библиотеку OkHttp.

ПРИМЕЧАНИЕ Не забудьте удалить лишние зависимости, чтобы не утяжелять будущую сборку.

В commonMain описана `expect`-декларация нашего класса сетевого клиента (листинг 13.2).

Листинг 13.2. Expect-декларация сетевого клиента

```
// Enum для методов клиента
public enum class Method {
    GET,
    POST,
    PUT,
    DELETE
}

// Декларация сетевого клиента
public expect class NetworkHttp constructor() {
    suspend inline fun <reified T> request(
        path: String,
        method: Method,
        body: Any? = null, headers: Map<Any?, *>
    ): Result<T>
}
```

`Actual`-реализации клиента для iOS и Android располагаем в iOSMain и androidMain соответственно.

Сетевой клиент для iOS реализован на API Network Framework iOS SDK, но в адаптации под Kotlin Native (листинг 13.3).

Листинг 13.3. Сетевой клиент для iOSMain на Kotlin Native

```
actual class NetworkHttp actual constructor() {

  val urlSession: NSURLSession by lazy {
    NSURLSession.sharedSession
  }

  actual suspend inline fun<reified T> request(
path: String,
      method: Method,
      body: Any?,
headers: Map<Any?, *>): Result<T> {

    val url = NSURL.URLWithString(path)
    val urlRequest = NSMutableURLRequest(url!!)
    urlRequest.setAllHTTPHeaderFields(headers)

    return suspendCoroutine<Result<T>> { coroutine ->
      urlSession.dataTaskWithRequest(urlRequest) {
          data,
```

```
          response,
          error ->

        try {
          val content = NSString.create(data!!,
              encoding =  NSUTF8StringEncoding).toString()
          coroutine.resume(Result.success(
              JsonDecoder
                  .instance.decode<T>(content))
          )
        } catch (e: Exception) {
          coroutine.resume(Result.failure(e))
        }
      }.resume()
    }
  }
}
```

Подробно его код мы разбирали в главе 5 о сетевом взаимодействии.

Сетевой клиент для Android реализуем с помощью библиотеки OkHttp (листинг 13.4).

Листинг 13.4. Сетевой клиент для Android

```
actual class NetworkHttp {
    val client: OkHttpClient by lazy {
        OkHttpClient()
    }

    actual suspend inline fun <reified T> request(
        path: String,
        method: Method,
        body: Any?, headers: Map<Any?, *>
    ): Result<T> {
        val requestBuilder = Request.Builder()
            .url(path)
        for (header in headers) {
            requestBuilder.addHeader(
                header.key.toString(),
                header.value.toString()
            )
        }

        requestBuilder.method(method.name, null)
        val request = requestBuilder.build()
        try {
            val response = client.newCall(request).await()
            val content = JsonDecoder
                .instance
                .decode<T>(
```

```
                    response.body!!.string()
                )
            return Result.success(content)
        } catch (e: Exception) {
            return Result.failure(e)
        }
    }
}
```

В примере мы используем упрощенные реализации сетевых клиентов. Если вы хотите доработать его и дополнить расширенным функционалом, вернитесь к главе 5 этой книги.

Для сетевого клиента Android мы также используем самописную обертку, чтобы преобразовать функции обратного вызова в корутины (листинг 13.5).

Листинг 13.5. Обертка для преобразования блоков обратного вызова в корутины

```
/** @see Call.await */
class ContinuationCallback(
    private val call: Call,
    private val continuation: CancellableContinuation<Response>
) : Callback, CompletionHandler {

    override fun onResponse(call: Call, response: Response) {
        continuation.resume(response)
    }

    override fun onFailure(call: Call, e: IOException) {
        if (!call.isCanceled()) {
            continuation.resumeWithException(e)
        }
    }

    override fun invoke(cause: Throwable?) {
        try {
            call.cancel()
        } catch (_: Throwable) {}
    }
}

suspend inline fun Call.await(): Response {
    return suspendCancellableCoroutine { continuation ->
        val callback = ContinuationCallback(this, continuation)
        enqueue(callback)
        continuation.invokeOnCancellation(callback)
    }
}
```

Для упрощения десериализации вынесем работу с `kotlinx.serialization` в специальный класс `JsonDecoder` (листинг 13.6).

Листинг 13.6. Пример JsonDecoder

```
class JsonDecoder{

    companion object {
        val instance = JsonDecoder()
        val jsonDecoder = kotlinx.serialization.json.Json { ignoreUnknownKeys =
true }
    }

    inline fun<reified T> decode(json:String): T {
        return jsonDecoder.decodeFromString(json)
    }
}
```

Готово. Мы перенесли весь код, который нужен для нашей библиотеки. Теперь перейдем к сборке и дистрибуции.

13.3. Сборка библиотеки

Для того чтобы мы могли распространять и подключать нашу библиотеку, необходимо собрать артефакты. Сделать это можно следующими способами:

- собрать артефакты через соответствующую команду assembleRealease;
- собрать Maven-артефакт локально;
- собрать и опубликовать Maven-артефакт в репозитории.

Первый способ позволит нам подключать артефакты библиотеки напрямую к нативным приложениям. Опубликовав в Maven, мы сможем подключить сборку к shared-модулю приложения Kotlin Multiplatform.

13.3.1. Сборка артефактов

В стандартном приложении KMP при запуске сборки из shared-модуля мы получаем артефакты для Android- и iOS-нативных приложений, которые автоматически подключаются к таргетам как библиотеки/фреймворки. Команды assemble позволяют подготовить артефакты, которые мы сможем подключить вручную.

Начнем со сборки артефакта для Android-таргета. Нам не нужно предварительно готовить библиотеку или ее код, но стоит убрать все лишние зависимости, чтобы не утяжелять готовый aar (Android Archive). Aar генерируется для Android-таргетов вместо jar.

Откроем терминал в каталоге нашего проекта или терминал в Android Studio и запустим команду:

```
./gradlew assembleRelease
```

В процессе сборки у нас генерируется артефакт `shared-release.aar` внутри папки build (рис. 13.4).

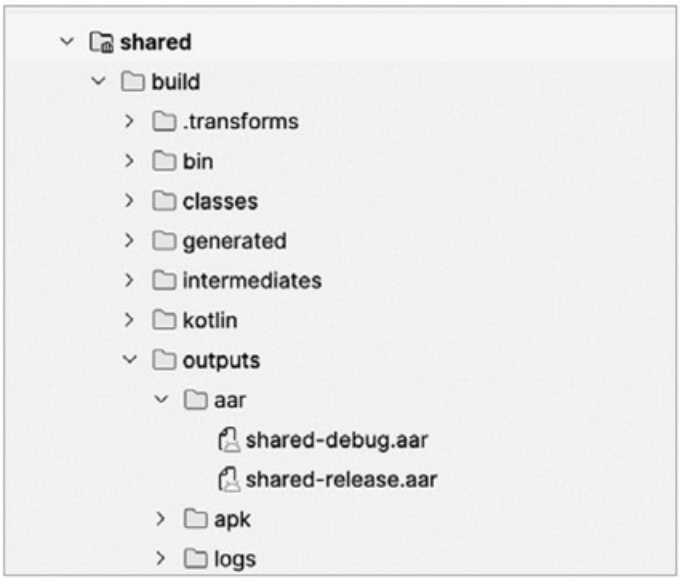


Рис. 13.4. Результат сборки

Для создания артефакта iOS потребуется внести следующие изменения в `build.gradle.kts` (листинг 13.7).

Листинг 13.7. Поддержка сборки в XCFramework

```
val xcf = XCFramework()
        listOf(
            iosX64(),
            iosArm64(),
            iosSimulatorArm64()
        ).forEach {
            it.binaries.framework {
                baseName = "shared"
                xcf.add(this)
                isStatic = true
            }
        }
```

Так наша библиотека соберется в формат XCFramework с поддержкой всех iOS-таргетов.

Запустим сборку артефакта с помощью команды:

```
./gradlew assembleXCFramework
```

Получим результат сборки, как на рис. 13.5.

Наш артефакт будет доступен в build ▸ XCFrameworks ▸ release.

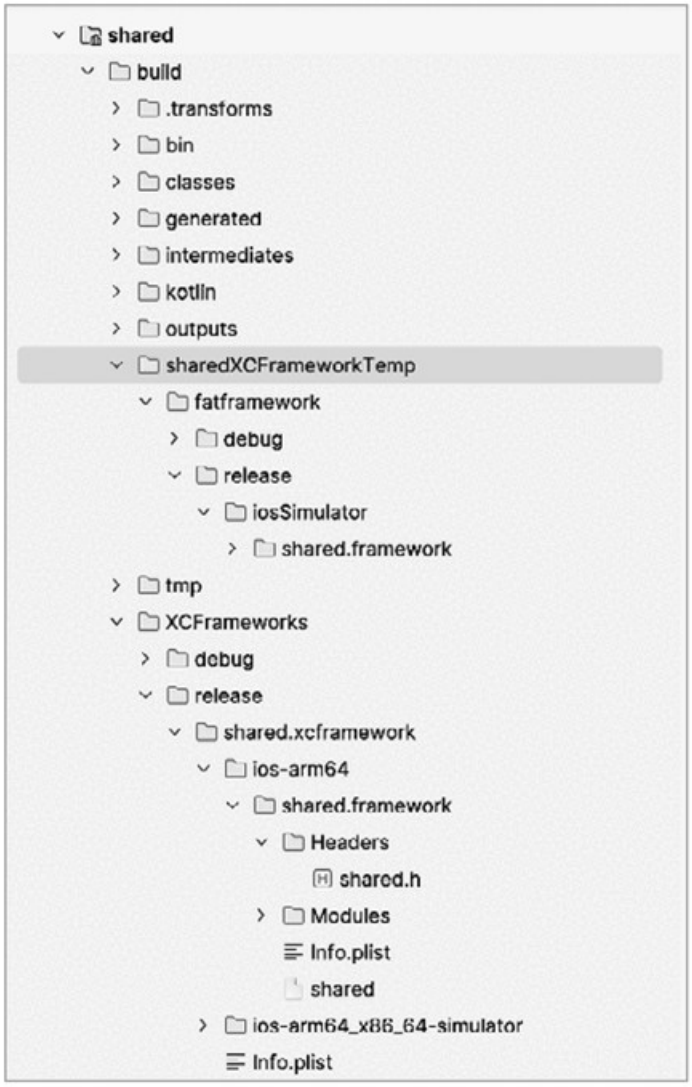


Рис. 13.5. Результат сборки для iOS

13.3.2. Подключение к iOS-приложению

Попробуем подключить собранные артефакты к тестовым приложениям и проверить.

Скопируйте `shared.xcframework` в ваше приложение в папку Frameworks. На вкладке Build Phases в разделе Link Binary with Libraries добавьте файл фреймворка (рис. 13.6).

На вкладке General в разделе Framework, Libraries and embedded content также добавьте библиотеку и установите ей статус Embed & sign.

Выполните очистку проекта с помощью Product ▸ Clean Build folder, а затем сборку с помощью Product ▸ Build.

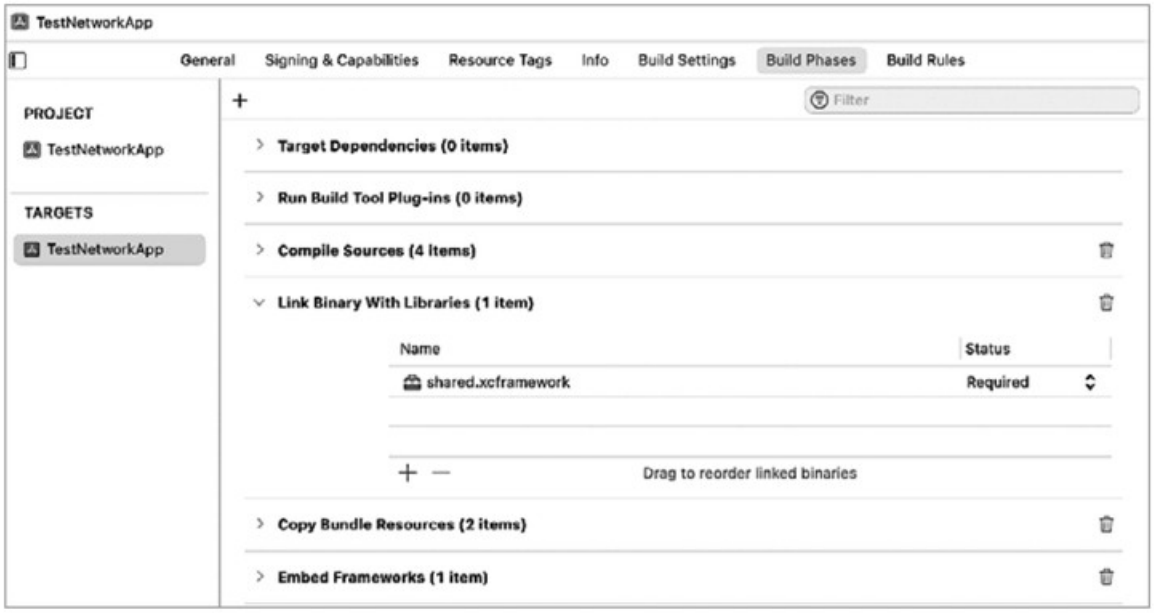


Рис. 13.6. Подключение XCFramework к проекту

Еще одним способом подключения артефакта XCFramework к приложению является использование пакетов Swift Package Manager.

Создадим новый пакет через Xcode File ▸ New ▸ Package (рис. 13.7).

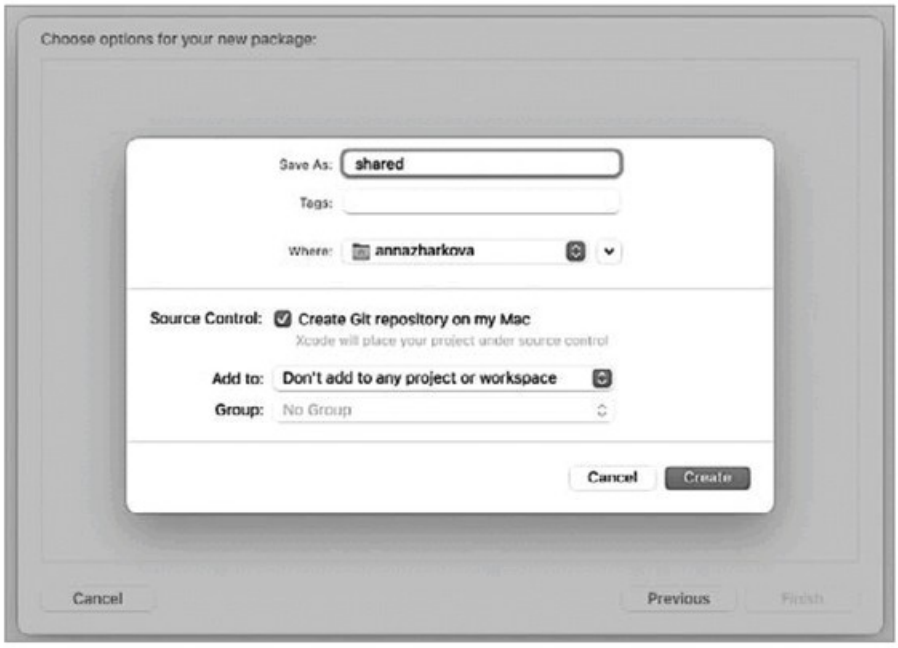


Рис. 13.7. Создание нового пакета

Желательно, чтобы имя пакета совпадало с именем `xcframework`.

Скопируем `shared.xcframework` в пакет и расположим на одном уровне с файлом `Package.swift` (рис. 13.8).



Рис. 13.8. Структура пакета

Путь к файлу фреймворка нужно прописать в `Package.swift` (листинг 13.8), задекларировав его как `binaryTarget`.

Листинг 13.8. Пакет библиотеки

```
import PackageDescription

let package = Package(
  name: "shared",
  platforms: [
    .iOS(.v14)
  ],
  products: [
    .library(
      name: "shared",
      targets: ["shared"])
  ],
  targets: [
    .binaryTarget(name: "shared", path: "./shared.xcframework")
  ]
)
```

Наш пакет готов. Добавим его к нужному проекту как локальный пакет (рис. 13.9).



Рис. 13.9. Добавление локального пакета к проекту

Теперь можно подключать библиотеку в код приложения. В качестве примера мы добавим вызов сетевого клиента из Observable Object нашего View (листинг 13.9).

Листинг 13.9. Пример вызова кода из подключенной библиотеки

```
class TestVM: ObservableObject {

  lazy var networkClient: NetworkHttp = {
    return NetworkHttp()
  }()

  func loadData() {
    networkClient.request<NewsList>(
      path: "https://newsapi.org/v2/everything?q=music",
      method: Method.get, body: nil,
      headers: ["X-Api-Key": "YOUR_KEY"]
    ) { data, error in
      if let data = data {
        print(data)
      }
    }
  }
}
```

Запустив код, мы получим ошибку (листинг 13.10).

Листинг 13.10. Ошибка при неверной реализации кода в shared

```
Uncaught Kotlin exception: kotlin.IllegalStateException: unsupported call
of reified inlined function `ru.azharkova.nativeloaderlib.NetworkHttp.request`
```

Проблема в неподдерживаемых Generics, которые мы используем для удобного декодирования. Наш код будет работать при вызове из shared-модуля, но для прямого вызова со стороны iOS надо доработать его с учетом интеропа Kotlin — Swift.

Для упрощенной версии достаточно добавить методы, которые будут отдавать ответ в виде строки или байтов (листинг 13.11).

Листинг 13.11. Доработка общего метода (expect-декларация)

```
expect class NetworkHttp constructor() {
    suspend inline fun <reified T> request(
        path: String,
        method: Method,
        body: Any? = null, headers: Map<Any?, *>
    ): Result<T>

    suspend fun <T:Any> requestString(
        path: String,
        method: Method,
        body: Any? = null, headers: Map<Any?, *>
    ): Result<String>
}
```

`Actual`-реализация практически идентична предыдущей, за исключением вызова декодера.

Изменим код на стороне iOS и получим рабочий вызов сетевого клиента (листинг 13.12).

Листинг 13.12. Исправленный вызов метода библиотеки

```
func loadData() {
  networkClient.requestString(
    path: "https://newsapi.org/v2/everything?q=music", method: Method.get, body:
nil,
    headers: ["X-Api-Key": "YOUR_KEY"]
  ) { data, error in
    if let data = data {
      //Обрабатываем код
    }
  }
}
```

13.3.3. Подключение к приложению Android

Теперь займемся подключением библиотеки к нативному Android-приложению.

В тестовом проекте создадим папку app/libs, куда скопируем полученный `aar` (рис. 13.10).



Рис. 13.10. Подключение aar

В `build.gradle.kts` пропишем путь к библиотеке (листинг 13.13).

Листинг 13.13. Настройка пути к библиотеке в build.gradle.kts

```
dependencies {
  implementation(fileTree(mapOf("dir" to "libs",
    "include" to listOf("*.jar", "*.aar"))))
  implementation(files("libs/shared-release.aar"))
  //....
}
```

Синхронизируем Gradle.

Добавим `ViewModel` в тестовое приложение и пропишем обращение к сетевому клиенту (листинг 13.14).

Листинг 13.14. Примерный код вызова библиотеки для тестирования

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers.IO
import kotlinx.coroutines.launch
import ru.azharkova.nativeloaderlib.Method
import ru.azharkova.nativeloaderlib.NetworkHttp

class MainViewModel {
    private val networkClient = NetworkHttp()

    fun loadData() {
        CoroutineScope(IO).launch {
           val result = networkClient.request<NewsList>(
                "https://newsapi.org/v2/everything?q=music",
                Method.GET,
                headers = mapOf("X-Api-Key" to "YOUR_KEY"),
            )

            println(result.getOrNull().toString())
        }
    }
}
```

Получим ошибку: некоторые типы не найдены (листинг 13.15).

Листинг 13.15. Пример содержания ошибки (не найдены классы)

```
java.lang.NoClassDefFoundError: Failed resolution of:
                    Lokhttp3/Request$Builder;
    at ru.azharkova.testnetwork.MainViewModel$loadData$1.
                    invokeSuspend(MainViewModel.kt:29)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.
                    resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:108)
    at kotlinx.coroutines.internal.LimitedDispatcher$Worker.
                    run(LimitedDispatcher.kt:115)
    at kotlinx.coroutines.scheduling.TaskImpl.run(Tasks.kt:103)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.
                    runSafely(CoroutineScheduler.kt:584)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.
                    executeTask(CoroutineScheduler.kt:793)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.
                    runWorker(CoroutineScheduler.kt:697)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.
                    run(CoroutineScheduler.kt:684)
Suppressed: kotlinx.coroutines.internal.DiagnosticCoroutineContextException:
                    [StandaloneCoroutine{Cancelling}@465562c, Dispatchers.IO]
```

При сборке в `*.aar` не внедряются все прописанные зависимости, поэтому потребуется дополнительно установить их и в тестовое приложение.

13.4. Упаковка и публикация артефакта для Maven

При использовании публикации в Maven как способа упаковки и дистрибуции все нужные для функционирования библиотеки зависимости упаковываются внутрь.

Для начала нам нужно внести некоторые изменения в файл `build.gradle.kts` модуля библиотеки и подключить плагин maven-publish для поддержки публикации в Maven (листинг 13.16).

Листинг 13.16. Настройка библиотеки для публикации в Maven

```
plugins {
    //...
    id("maven-publish")
}
// Задаем версию и группу
group = "io.github.azharkova.nativeloaderlib"
version = "1.0.0"
```

Обязательно нужно задать группу и версию для создаваемого артефакта. Группа может быть связана с package вашей библиотеки либо с идентификаторами группы проектов/компании или аккаунтом на Github/Bitbucket.

Для Android-таргета нужно включить поддержку сборки под релизную конфигурацию или под все поддерживаемые (листинг 13.17).

Листинг 13.17. Настройка параметров сборки под конфигурацию Android

```
kotlin {
  androidTarget {
    publishLibraryVariants("release")
    //или
    publishLibraryVariantsGroupedByFlavor = true
    //..
  }
}
```

Для поддержки сборки под Kotlin/Native добавим в `gradle.properties` следующие значения (листинг 13.18).

Листинг 13.18. Флаги поддержки кросс-компиляции и синтеропа (cinterop)

```
kotlin.native.enableKlibsCrossCompilation=true
kotlin.mpp.enableCInteropCommonization=true
```

Последний флаг нужен для поддержки cinterop. Синхронизируем Gradle. Для сборки запускаем в терминале команду:

```
./gradlew :shared:publishToMavenLocal
```

В каталоге локального maven /.m2 внутри домашней папки появятся сгенерированные артефакты (рис. 13.11).

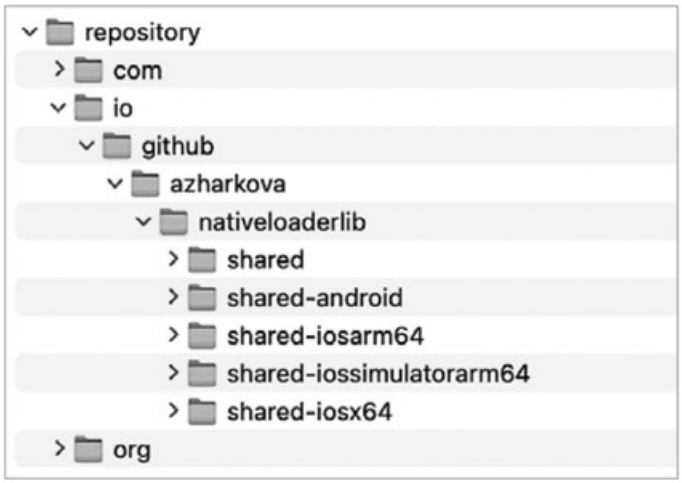


Рис. 13.11. Сгенерированные артефакты

Для iOS у нас версии артефактов собираются под каждую поддерживаемую архитектуру.

Проверим работу нашей библиотеки: создадим тестовое приложение Kotlin Multiplatform и добавим артефакт как зависимость (листинг 13.19).

Листинг 13.19. Подключение библиотеки KMP к тестовому приложению

```
sourceSets {
    commonMain.dependencies {
        implementation(libs.kotlinx.serialization.json)
        implementation("io.github.azharkova.nativeloaderlib:shared:1.0.0")
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.1")
    }
    androidMain.dependencies {
        implementation("io.github.azharkova.nativeloaderlib:shared-
                        android:1.0.0")
    }
    iosMain.dependencies {
        implementation("io.github.azharkova.nativeloaderlib:shared:1.0.0")
    }
}
```

Для Android мы добавим зависимость shared-Android из группы nativeloaderlib, для commonMain и iOSMain — просто shared из этой же группы.

Дополнительно добавим зависимости для корутин и сериализации, но только потому, что этот функционал мы используем уже в общем модуле тестового приложения в моделях данных и для асинхронных вызовов. Подключенные внутри нашей библиотеки зависимости доступны только ее функционалу и изолированы от нас, как в песочнице.

Для того чтобы наши зависимости могли корректно загрузиться, нужно указать `mavenLocal` в списке репозиториев в `settings.gradle` (листинг 13.20).

Листинг 13.20. Подключение доступа к локальному Maven

```
pluginManagement {
  repositories {
    google()
    gradlePluginPortal()
    mavenCentral()
    mavenLocal()
  }
}

dependencyResolutionManagement {
  repositories {
    google()
    mavenCentral()
    mavenLocal()
  }
}
```

Синхронизируем Gradle и проверим работу кода.

Если вы посмотрите на раздел External Libraries в иерархии проекта, то заметите, что все наши артефакты установились (рис. 13.12).

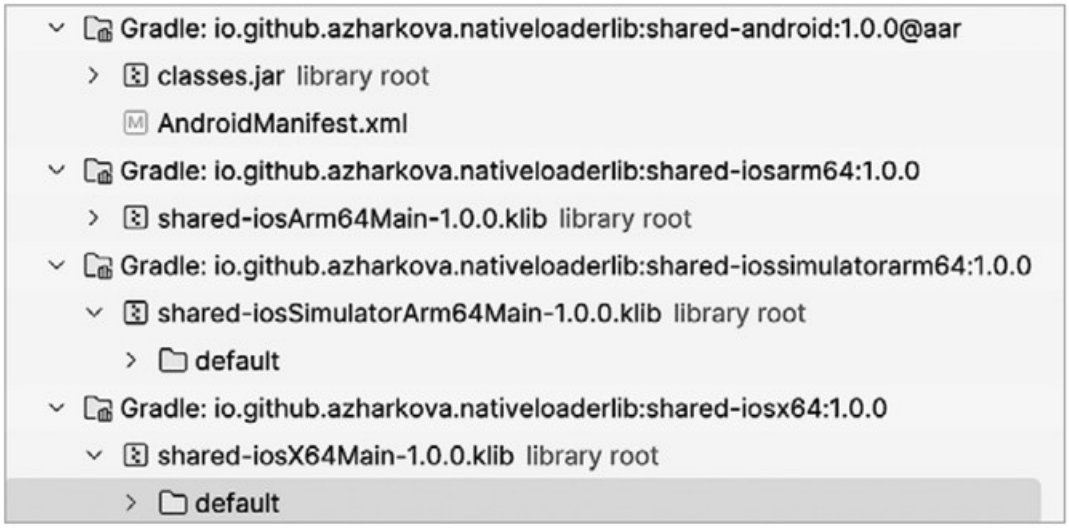


Рис. 13.12. Установленные артефакты библиотеки в External Libraries

13.5. Размещение библиотеки в публичном репозитории

После того как все доработки завершены, можно попробовать разместить библиотеку в публичном репозитории.

Мы уже рассмотрели способ публикации артефактов для iOS через Swift Package Manager локально. Этот же SPM можно будет использовать для удаленного репозитория.

13.5.1. Github Packages

В качестве примера публикации кросс-платформенной библиотеки рассмотрим работу с Github Packages.

На первом этапе нужно создать репозиторий для размещения пакетов и библиотеки. Пример публикации библиотеки будет расположен по адресу https://github.com/anioutkazharkova/native_network_lib.

Добавьте в `build.gradle.kts` модуля библиотеки код из листинга 13.21.

Листинг 13.21. Настройка поддержки публикации приложения в GithubPackages

```
publishing {
   repositories {
       maven {
           name = "GitHubPackages"
           url = uri("https://maven.pkg.github.com/username/repoName")
           credentials {
               username = providers.environmentVariable("GITHUB_USER").get()
               password = providers.environmentVariable("GITHUB_TOKEN").get()
           }
       }
   }

}
```

Мы задаем публикацию артефактов нашей библиотеки в `GithubPackages` через Maven. Замените `username` и `repoName` на ваши данные. Имя пользователя и токен для доступа к Github стоит указать через переменные среды либо свойства `local.properties`. Затем нужно вызвать в терминале следующую команду (листинг 13.22).

Листинг 13.22. Команды для публикации в GithubPackages

```
./gradlew publishAllPublicationsToGitHubPackagesRepository
    publishAndroidReleasePublicationToGitHubPackagesRepository
    publishKotlinMultiplatformPublicationToGitHubPackagesRepository
```

Проверим результат (рис. 13.13).

Для того чтобы использовать опубликованную библиотеку, добавьте `maven.pkg.github.com` в доступные репозитории:

```
repositories {
   maven {
       url = https://maven.pkg.github.com/user/repoName"
credentials {}
   }
}
```



Рис. 13.13. Опубликованные артефакты

Необходимо указать логин юзера и токен для доступа к репозиторию, поэтому такой способ не подойдет для некорпоративных проектов.

13.5.2. Публикация в Maven Central

Для публикации в Maven Central рекомендуем ознакомиться с туториалом в документации KotlinMultiplatform: https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-publish-libraries.html.

Зарегистрируйтесь на портале https://central.sonatype.com/.

1. В разделе `View Namespaces` вашей учетной записи выберите `Add Namespace` для добавления пространства имен.

   Пространство имен обязательно должно пройти верификацию, поэтому самым простым будет создать его по одному из паттернов:

   - Github: io.github.myusername;
   - Gitlab: io.gitlab.myusername;
   - Gitee: io.gitee.myusername;
   - Bitbucket: io.bitbucket.myusername.

   Выберите то пространство, где у вас есть учетная запись, так как для верификации потребуется создать проект (рис. 13.14).

2. На следующем этапе необходимо создать ключ для защиты данных.

   Установите утилиту gpg: `brew install gnupg`.

   Создайте ключ с помощью команды:

```
gpg --generate-key
```

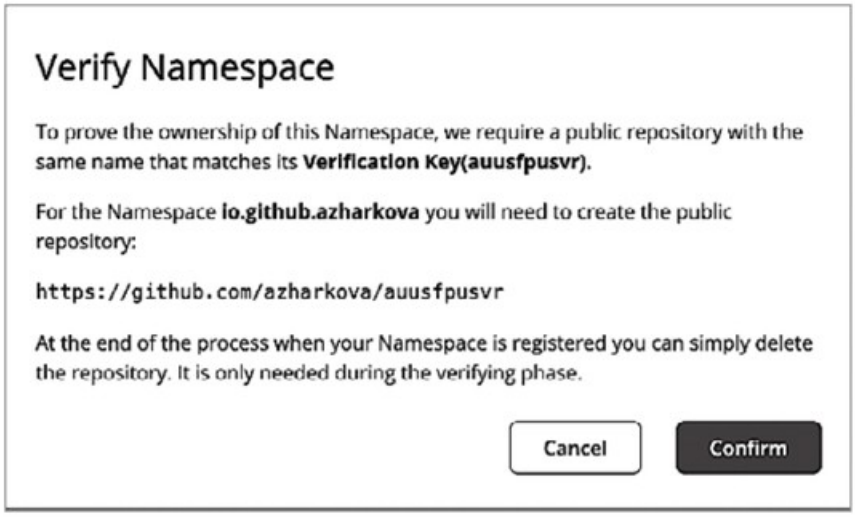


Рис. 13.14. Верификация пространства имен

Полученный ключ необходимо будет опубликовать на специальном сервере:

```
gpg --keyserver keyserver.ubuntu.com --send-keys YOUR_KEY
```

Этот этап может потребовать времени, так как серверы часто бывают перегружены.

3. Настройте проект.

   Для упрощения публикации в Maven в документации рекомендуется использовать специальный плагин:

```
plugins {
    id("com.vanniktech.maven.publish") version "0.30.0"
}
```

Добавим секцию в `build.gradle.kts` для настройки параметров публикации (листинг 13.23).

Листинг 13.23. Примерный блок настройки параметров публикации Maven

```
mavenPublishing {
    publishToMavenCentral(SonatypeHost.CENTRAL_PORTAL)

    signAllPublications()

    coordinates("io.github.anioutkazharkova", "shared", "1.0.1")

    pom {
        name = "NativeLoaderLib"
        description = "Small Kotlin Multiplatform Library for native network"
        inceptionYear = "2025"
        url = "https://github.com/anioutkazharkova/native_network_lib"
        licenses {
            license {
```

```
                name = "The Apache License, Version 2.0"
                url = "https://www.apache.org/licenses/LICENSE-2.0.txt"
                distribution = "https://www.apache.org/licenses/
                                                    LICENSE-2.0.txt"
            }
        }
        developers {
            developer {
                id = "anioutkazharkova"
                name = "Anna Zharkova"
                url = "https://github.com/anioutkazharkova/"
            }
        }
        scm {
            url = "https://github.com/anioutkazharkova/native_network_lib"
            connection = "scm:git:git://anioutkazharkova/
                                        native_network_lib.git"
            developerConnection = "scm:git:ssh://git@github.com/
                                  anioutkazharkova/native_network_lib.git"
        }
    }
```

Через `coordinates` нужно задать ваше пространство имен, название артефакта (название общего модуля для нашей библиотеки) и версию.

ВАЖНО! Группа вашей библиотеки должна совпадать с пространством имен.

Вам потребуется заменить своими данными адреса репозитория (https и git), имя пользователя и другие данные аккаунта.

4. Сгенерируйте токен для Sonatype.

   На сайте откройте раздел View Accounts, нажмите на Generate User Token и скопируйте полученные данные (листинг 13.24).

Листинг 13.24. Данные для генерации токена

```
<server>
    <id>${server}</id>
    <username>account</username>
    <password>password</password>
</server>
```

5. Настройте секретные переменные в гит-репозитории.

   Зайдите в раздел настроек вашего репозитория и откройте Secrets and variables. Нужно прописать следующие переменные:

   - `GPG_KEY_CONTENTS` — содержимое вашего gpg-ключа;
   - `MAVEN_CENTRAL_PASSWORD` — пароль для токена, полученный на предыдущем этапе;
   - `MAVEN_CENTRAL_USERNAME` — имя аккаунта для токена;

- SIGNING_KEY_ID — последние 8 знаков идентификатора вашего gpg-ключа;
- SIGNING_PASSWORD — пароль к вашему gpg-ключу.

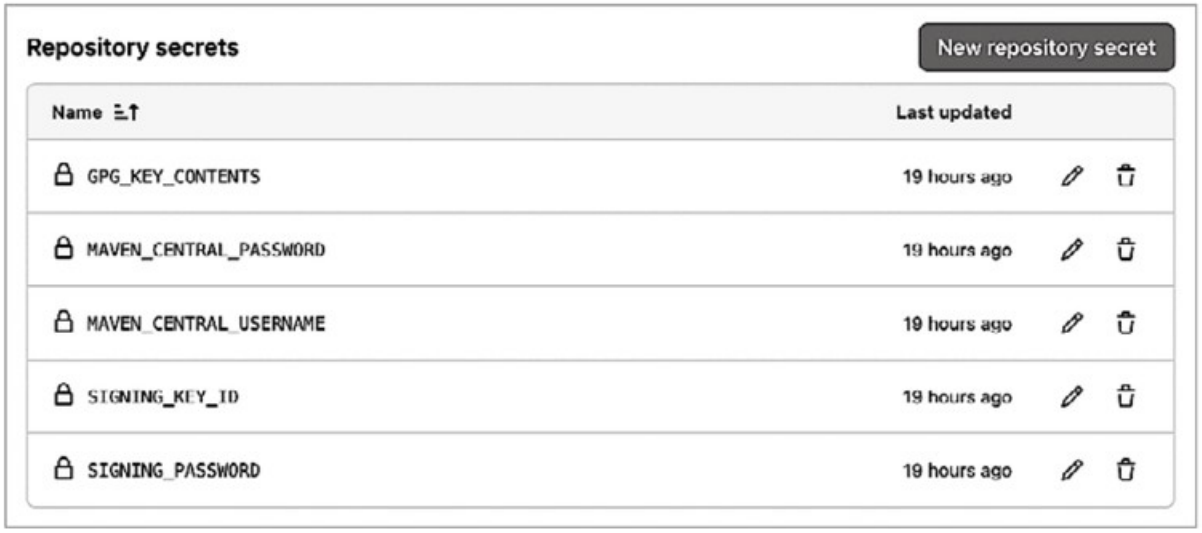


Рис. 13.15. Секретные переменные

6. Настройте Github Actions WorkFlow для сборки проекта.

   В документации предлагается использовать механизм Github Actions для сборки и загрузки артефакта в Maven Central.

   Предварительно необходимо загрузить код вашей библиотеки в репозиторий.

   Зайдите в раздел Github Actions и запустите создание нового WorkFlow. Скопируйте и вставьте в файл .yml примерно такой код (листинг 13.25):

Листинг 13.25. Publish.yml

```
# .github/workflows/publish.yml

name: Publish
on:
  release:
    types: [released, prereleased]
jobs:
  publish:
    name: Release build and publish
    runs-on: macOS-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v4
      - name: Set up JDK 21
        uses: actions/setup-java@v4
        with:
          distribution: 'zulu'
          java-version: 21
      - name: Publish to MavenCentral
        run: ./gradlew publishToMavenCentral --no-configuration-cache
```

```
env:
  ORG_GRADLE_PROJECT_mavenCentralUsername: ${{ secrets.MAVEN_CENTRAL_
            USERNAME }}
  ORG_GRADLE_PROJECT_mavenCentralPassword: ${{ secrets.MAVEN_CENTRAL_
            PASSWORD }}
  ORG_GRADLE_PROJECT_signingInMemoryKeyId: ${{ secrets.SIGNING_KEY_ID }}
  ORG_GRADLE_PROJECT_signingInMemoryKeyPassword: ${{ secrets.SIGNING_
            PASSWORD }}
  ORG_GRADLE_PROJECT_signingInMemoryKey: ${{ secrets.GPG_KEY_CONTENTS }}
```

После подготовки и проверки параметров среды окружения этот скрипт запускает команду публикации с использованием указанных секретных переменных.

7. Создайте релиз.

   Для того чтобы запустить сборку, нужно создать новый релиз нашей библиотеки. Зайдите в раздел Releases и выберите Draft new release/Create new release. Заполните поля и нажмите Publish (рис. 13.16).



Рис. 13.16. Создание нового релиза

ВАЖНО! Обязательно создайте метку-тег на вашем репозитории для обозначения версии кода, которая будет использована при сборке.

После публикации релиза сборка запускается автоматически.

8. Опубликуйте в SonaType.

   Как только WorkFlow на вкладке Actions завершится успешно, на сайте SonaType в вашем кабинете в разделе ViewDeployments появится информация о загруженной сборке. Если процесс пройдет без ошибок, то сборка будет в стадии Validated (рис. 13.17).



Рис. 13.17. Статус загруженной сборки

Нажмите на кнопку Publish для публикации артефакта в Maven Central. Спустя 10–15 минут сборка из статуса Validated перейдет в стадию Published (рис. 13.18).



Рис. 13.18. Опубликованный артефакт

Это значит, что библиотека готова к использованию для всех.

ПОДВЕДЕМ ИТОГИ

- ✓ Чтобы использовать библиотеку напрямую с нативными приложениями, потребуется собрать артефакты через assembleRelease и подключить их вручную к проектам.
- ✓ Для iOS возможно упростить работу, портируя `xcframework` через SPM. Этот же подход подойдет для широкого распространения компонента.
- ✓ Сборка артефакта Maven для Android позволяет встраивать зависимости прямо в библиотеку.
- ✓ Для широкого распространения артефактов используйте готовые сервисы: Github Packages, Maven Central.

ГЛАВА 14

Гайд-шпаргалка по миграции существующих нативных приложений

В этой главе рассмотрим:

- миграцию существующего Android-приложения на Kotlin Multiplatform;
- миграцию существующего iOS-приложения на Kotlin Multiplatform.

В этой книге мы последовательно в каждой главе разобрали, как создать кросс-платформенное приложение и настроить и адаптировать нативные таргеты.

Ниже представлено итоговое резюме проделанной нами работы в виде дорожной карты разработчика KMP по миграции нативного приложения.

14.1. Из Android в Kotlin Multiplatform

1. Создайте новый модуль приложения через меню File ▸ New Module ▸ Kotlin Multiplatform Shared Module (рис. 14.1).
2. Подключите новый модуль к Android-приложению как зависимость в `build.gradle.kts` (листинг 14.1).

Листинг 14.1. Подключение shared-модуля к нативному приложению

```
dependencies {
  implementation(projects.shared)
  //..
}
```

   Запустите синхронизацию.

3. Начните перенос логики. Подключите библиотеки для сериализации `Kotlinx.serialization` и корутин (листинг 14.2).

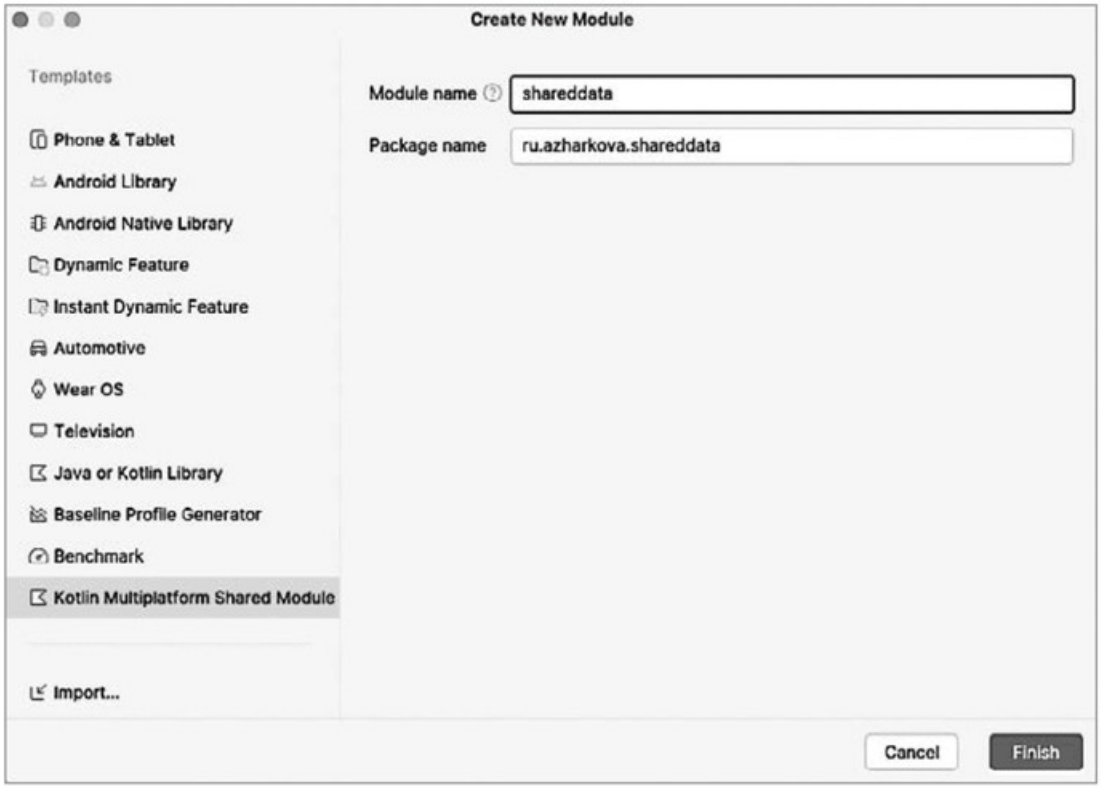


Рис. 14.1. Добавление нового модуля

Листинг 14.2. Подключение библиотеки и корутин

```
//lib.versions
[versions]
kotlinxSerializationJson = "1.8.0"
kotlinSerialization = "2.1.0"
coroutines = "1.10.1"

[libraries]
kotlinx-serialization-json = { module = "org.jetbrains.kotlinx:kotlinx-
        serialization-json", version.ref = "kotlinxSerializationJson" }
coroutines = {module = "org.jetbrains.kotlinx:kotlinx-coroutines-core",
        version.ref = "coroutines"}


[plugins]
serialization = {id=  "org.jetbrains.kotlin.plugin.serialization",
        version.ref = "kotlinSerialization"}

//build.gradle.kts
sourceSets {
     commonMain.dependencies {
     //put your multiplatform dependencies here
     implementation(libs.coroutines)
     implementation(libs.kotlinx.serialization.json)
  }
}
```

4. Перенесите модели данных в shared-модуль. Замените аннотацию `java.io.Serializable` на `kotlinx.serialization.Serializable` (листинг 14.3–14.4).

Листинг 14.3. До миграции

```
import java.io.Serializable

data class NewsItem(
        val source: Source? = null,
        @Expose
        @Serialized("author")
        val author: String?,
        val title: String,
        val description: String,
        val url: String? = null,
        val publishedAt: String? = null,
        val content: String,
        val urlToImage: String? = null
) : Serializable
```

Листинг 14.4. После миграции

```
@kotlinx.serialization.Serializable
data class NewsItem(
        val source: Source? = null,
        @SerialName("author") val author: String?,
        val title: String,
        val description: String,
        val url: String? = null,
        val publishedAt: String? = null,
        val content: String,
        val urlToImage: String? = null
)
```

На этом этапе нужно заменить в нативном приложении ссылки на модели данных.

5. Для работы с многопоточностью создайте общие диспетчеры корутин (листинг 14.5).

Листинг 14.5. Диспетчеры корутин

```
val uiDispatcher: CoroutineDispatcher = Dispatchers.Main

val ioDispatcher: CoroutineDispatcher = Dispatchers.IO

val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
```

6. Реализуйте сетевой клиент на Ktor и замените им клиент на Retrofit (см. главу 5).

   Новый сетевой клиент временно можно зарегистрировать в DI Android-приложения и подключить к сервису `ViewModel`.

7. Перенесите хранилища с `SharedPreferences` на DataStore (см. главу 8). Если у вас есть локальная база данных на Room, перед переносом проанализируйте, есть ли у вас важный функционал, который не поддерживается Kotlin Multiplatform.
8. Перенесите сервисы `useCase`, которые используются в Android-приложении, в shared-модуль.
9. Замените DI Android-приложения на Koin/Kotlin Inject. Компонент реализуйте в общем модуле (см. главу 7). Если вы используете Dagger/Hilt, то, возможно, будет удобнее мигрировать с него на DI для Kotlin Multiplatform раньше. Порядок этого шага выбирайте в зависимости от вашего проекта.
10. Перенесите `ViewModel` Android в shared-модуль и замените версию библиотеки на поддерживающую кросс-платформу (листинг 14.6).

Листинг 14.6. Библиотека для ViewModel

```
commonMain.dependencies {
    implementation(libs.androidx.lifecycle.viewmodel)
}
```

11. Если у вас в проекте используется архитектура, отличная от MVVM, то можно перенести в общий модуль все, кроме UI (листинг 14.7).

Листинг 14.7. Конфигурация модуля экранной логики

```
class MoviesListConfigurator : IConfigurator {
    companion object {
        val instance = MoviesListConfigurator()
    }

    override fun create(view: IView): IInteractor? {
        val interactor: IMoviesListInteractor = MoviesListInteractor()
        val presenter = MoviesListPresenter()
        interactor.presenter = presenter
        presenter.view = view as? IMoviesListView
        return interactor
    }
}
```

Фабрику конфигураторов тоже можно перенести в общий модуль (листинг 14.8).

Листинг 14.8. Фабрика конфигураторов

```
interface IConfigurator {
    fun create(view: IView):IInteractor?
}

class ModuleConfig {
    companion object {
```

```
            val instance = ModuleConfig()
        }

        fun config(view: IView):IInteractor? {
             if (view is IMoviesListView) {
                 return MoviesListConfigurator.instance.create(view)
             }
            return null
        }
    }
```

ВНИМАНИЕ! Если у вас UI на Compose, то вам потребуется модель состояния как связующее звено между View и остальными компонентами экранной логики.

12. На завершающем этапе перенесите UI. Если вы до сих пор используете подход xml, то ознакомьтесь с главой 10. Если UI у вас реализован на Compose, то установите в commonMain зависимости для Compose Multiplatform (листинг 14.9).

Листинг 14.9. Установка зависимостей для Compose Multiplatform

```
commonMain.dependencies {

    implementation(compose.ui)
    implementation(compose.runtime)
    implementation(compose.foundation)
    implementation(compose.material3)
    implementation(compose.components.resources)
    implementation(libs.kotlin.test)
    api("moe.tlaster:precompose:1.5.7")
    implementation(compose.components.resources)
    api("io.github.qdsfdhvh:image-loader:1.7.1")
}
```

13. Перенесите ваши Composable в общий модуль и обновите import.
14. Реализуйте легковесную навигацию на PreCompose (см. главу 10).
15. При необходимости и по желанию покройте приложение тестами (см. главу 9).

Готово! Теперь у вас полностью кросс-платформенное приложение. При желании вы можете мигрировать не полностью, а только некоторыми слоями или компонентами.

14.2. Миграция существующего iOS-приложения

Во многом миграция с iOS на KMP равносильна разработке приложения с нуля. Нативное приложение выполняет роль базы и ориентира для сохранения логики.

1. Если у вас нет готового кросс-платформенного приложения с нативным Android-таргетом, то создайте новый проект кросс-платформенной библиотеки (рис. 14.2).

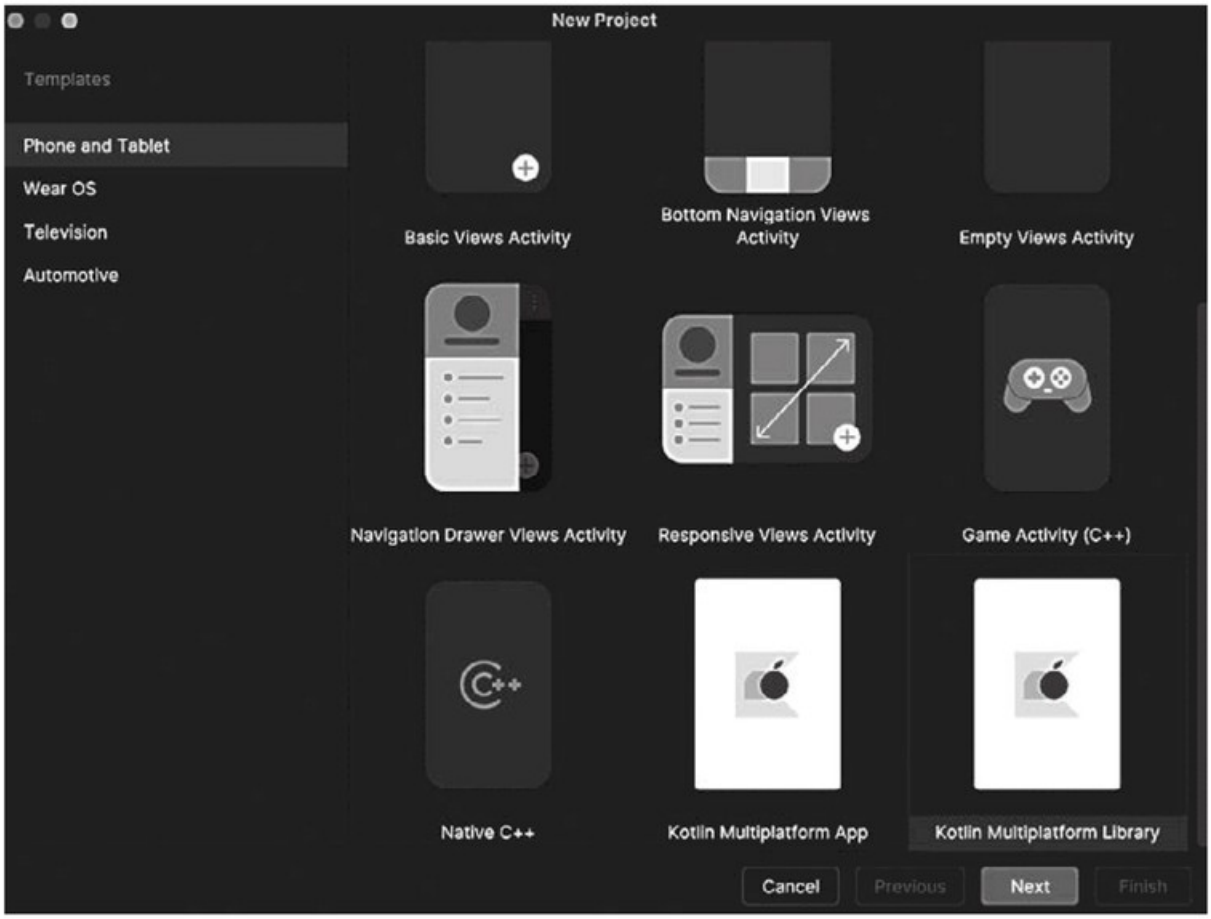


Рис. 14.2. Создание пустого shared-модуля

2. В нативном iOS-проекте на вкладке Build Settings в Search Path укажите путь для поиска нашего собранного фреймворка:

```
$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)
```

Потом в Linking ▸ General в Other linking flags добавьте название модуля (в нашем случае shared) и флаги `$(inherited)` и `-frameworks` (рис. 14.3).

Чтобы shared-модуль автоматически собирался в `xcodeframework`, необходимо прописать скрипт. На вкладке Build Phases (рис. 14.4) добавьте Run script с кодом:

```
cd "$SRCROOT/.."
./gradlew :shared:embedAndSignAppleFrameworkForXcode
```

3. Начинаем пошаговую миграцию. Подключите библиотеки для сериализации и многопоточности.

4. Создайте data-классы вместо классов и структур Swift. Замените ими нативные структуры данных.



Рис. 14.3. Параметры для iOS-нативного приложения

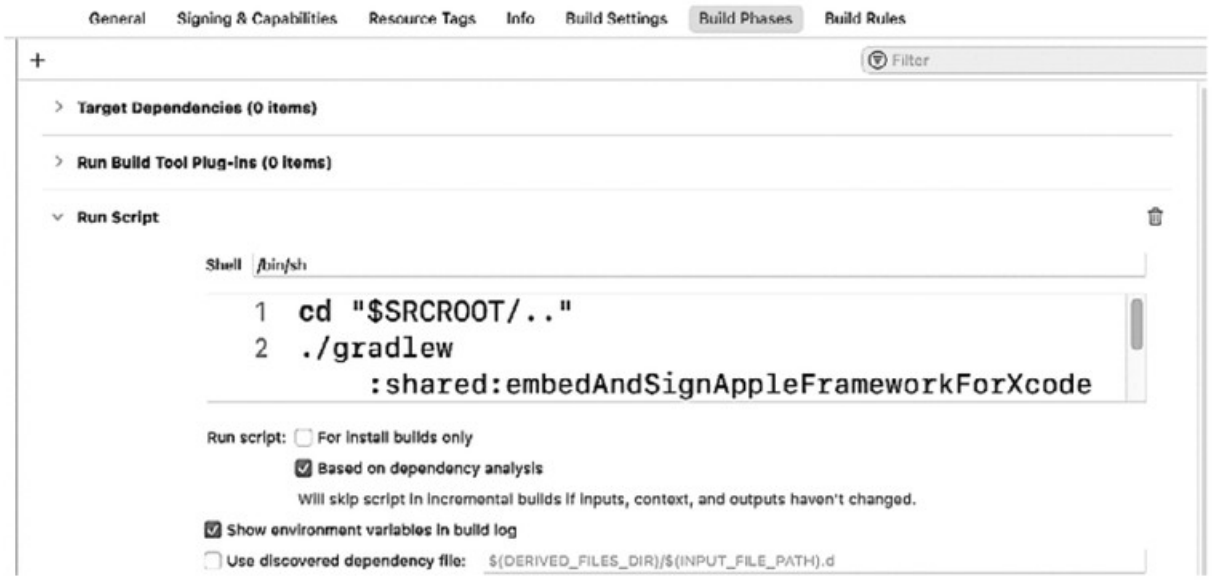


Рис. 14.4. Подключение скрипта в Build Phases

5. Добавьте диспетчеры корутин в shared-модуль (см. главу 4). При необходимости вызова Flow или корутин со стороны iOS-приложения реализуйте вспомогательные компоненты (листинг 14.10).

Листинг 14.10. Пример класса-обертки Flow

```
class AnyFlow <T> (source: Flow <T> ): Flow <T> by source {

  fun collect(onEach: (T) -> Unit,
```

```
    onCompletion: ((cause: Throwable ? )-> Unit)? = null): Cancellable {

    // Скоуп работает внутри метода
    val scope = CoroutineScope(SupervisorJob() + Dispatchers.Main)

    scope.launch {
      try {
        // Подписываемся на результат Flow
        collect {
          onEach(it)
        }
        // Если нет ошибки, то просто вызываем завершение
        onCompletion(null)
      } catch (e: Throwable) {
        onCompletion?.invoke(e)
      }
    }

    return object: Cancellable {

      // Зачищаем скоуп, убираем подписку
      override fun cancel() {
        scope.cancel()
      }
    }
  }
}

fun <T> Flow<T>.wrapToAny(): AnyFlow<T> = AnyFlow(this)
```

6. Реализуйте сетевой слой на Ktor (см. главу 5). Замените им сетевой слой на Network Framework, Moya/Alamofire. На этом этапе его можно вызывать напрямую на стороне нативного iOS-приложения.
7. Вместо хранилища на SwiftData настройте хранилище Room в shared-модуле (см. главу 8). Также можно временно использовать его напрямую в iOS-приложении.
8. Сервисы, `useCase` и менеджеры по возможности перенесите в общий модуль.
9. Следующим шагом будет адаптация архитектуры и настройка DI (см. главу 7). Исходите из особенностей вашего проекта и будущих потребностей. Повторите шаги 8 и 11 из раздела 14.1.

   Не забудьте, что для DI потребуется доработать специальный метод расширения для поддержки запроса инстансов типов через `ObjCClass`.

   ВНИМАНИЕ! Если вы оставляете нативный UI, то у вас останется и `ViewModel` ObservableObject как модель состояния.

10. Добавьте кросс-платформенную `ViewModel` в shared-модуль (глава 6). Доработайте связи между бизнес-логикой и архитектурными компонентами экранов.

11. Если у вас в проекте нет ограничений и особых требований по реализации UI, то вы можете также реализовать его в shared-модуле с помощью Compose Multiplatform (глава 10). В противном случае придется оставить нативный UI или подключить его к Compose через обертку.

    В качестве механизма навигации выберите PreCompose, Decompose для UI на Compose Multiplatform.

12. При желании покройте код тестами (глава 9).

    ВНИМАНИЕ! Если вы используете нативные таргеты (виджеты, LiveActivity, AppClips), в которых есть общий с вашим приложением iOS-код, учитывайте, что shared-модуль напрямую им не доступен. Потребуется делать класс-обертку для вызова общей логики. Также в некоторых кейсах эти таргеты используют SwiftData.

Готово! Вы мигрировали с iOS на Kotlin Multiplatform.

ПРИМЕЧАНИЕ Помните: даже если какие-то компоненты нельзя заменить на полностью кросс-платформенные или объединить с помощью expect/actual, можно попробовать объединить их с помощью интерфейсов или написать в androidMain/iOSMain.

Одним из основных критериев допустимости переноса кода в shared-модуль является сокращение избыточной связности кода.

Резюме

Поздравляем! Вы завершили свое путешествие по книге «Kotlin Multiplatform на практике», но не по миру кросс-платформенной разработки. Настоящая работа для вас только начинается.

Сначала вы познакомились с особенностями создания проекта с нуля и миграции существующего нативного приложения на кросс-платформу.

Потом вы рассмотрели специфику архитектуры приложений на KMP, работу с нативным UI, концепцию общего и нативного кода, а также различные случаи ее правильного применения в проектах и на практике. Также вы могли более глубоко погрузиться в работу с памятью и нюансы реализации многопоточности в Kotlin Multiplatform.

После этого вы перешли к работе с сетью и хранилищами данных в приложениях KMP, познакомились со спецификой нативных решений и готовых библиотек. Также вы узнали, как соединить слои кросс-платформенного приложения с помощью различных механизмов Dependency Injection.

Потом вы освоили продвинутые технологии и способы улучшить свое приложение и повысить качество кода; вы изучили, как работать с тестами кросс-платформенно, какие готовые решения для этой задачи существуют.

Затем вы узнали об особенностях работы с UI/UX в приложениях KMP, о различных способах создания и реализации UI, в том числе с помощью Compose Multiplatform. Также вы изучили способы организации навигации, готовые решения для работы с UI и совмещение нативного UI с общим.

После этого вы познакомились со средствами кодогенерации и особенностями их использования для ускорения разработки кросс-платформенного приложения.

Также вы узнали о создании, разработке и распространении собственных кросс-платформенных библиотек и их подключении к приложениям.

В завершение вы ознакомились с полной инструкцией по миграции ваших приложений на KMP.

По ходу изучения материала книги вы могли обратить внимание на специальные главы и разделы, посвященные изучению языков Kotlin и Swift, а также нюансам и особенностям Android и iOS SDK. Они позволили вам освоить необходимый минимум по второй платформе и углубить свои знания по той, на которой вы специализируетесь.

Также результатом вашего прочтения книги является готовый собственный проект, который вы разрабатывали по ходу разбора и погружения в материал. Если вы уже были знакомы с KMP ранее, то вам было полезно читать эту книгу как справочник по отдельным темам.

В любом случае вы углубили свои знания и освоили новые технологии.

Что дальше

Будем рады вашим отзывам, комментариям и предложениям. Присылайте их автору в Telegram-канал @prettygeeknotes. Код для примеров вы сможете найти по ссылке: https://gitverse.ru/azharkova/kmp_book.

Обязательно подпишитесь на канал автора и следите за новостями. Следующее, дополненное издание уже в работе! В нем, помимо прочего, мы рассмотрим разработку под таргеты Desktop, WASM, а также Aurora OS.

Перспективы Kotlin Multiplatform

Подробнее о дорожной карте развития KMP и планах разработчиков вы можете узнать из блога JetBrains: https://blog.jetbrains.com/kotlin/2024/10/kotlin-multiplatform-development-roadmap-for-2025/.

В настоящее время компания Google проявляет большую заинтересованность в развитии данной технологии и активно поддерживает ее создателей. Инженеры Google не только помогают портировать существующие решения для Android на KMP и продвигать подход Kotlin First. Kotlin Multiplatform также объявлен официально рекомендованным решением для разработки продвинутой и эффективной бизнес-логики для нативных приложений. Именно это обстоятельство делает технологию весьма перспективной в ближайшие годы.

В своем блоге компания JetBrains подробно описывает основные цели разработки на ближайшее время. Самыми ожидаемыми в 2025 году для большинства разработчиков являются:

- стабилизация Compose Multiplatform iOS. Это позволит создавать улучшенный общий UI и работать с ним более эффективно;
- прямой экспорт кода Kotlin в Swift без промежуточной конвертации в C/Objectvie-C. Это ускорит работу iOS-таргета, повысит производительность и оптимизирует управление памятью.

Также для разработчиков были бы полезны готовые решения по организации навигации и управлению состоянием приложения.

Все эти доработки ожидаемо повысят качество ваших приложений, хотя и не решат полностью все задачи, связанные с кросс-платформенной разработкой под разные таргеты. Ответы на оставшиеся вопросы вы найдете в этой книге.

Удачи и успехов в работе!

Анна Жаркова