ХАСБУЛАТОВ М. В., ИРЗАЕВ Г. Х. ПРАКТИЧЕСКИЕ АСПЕКТЫ ИНТЕГРАЦИИ KOTLIN MULTIPLATFORM В АРХИТЕКТУРУ ANDROID-ПРИЛОЖЕНИЯ

УДК 004.42, ГРНТИ 50.05.19

Статья поступила в редакцию 23.11.2025

Практические аспекты интеграции Kotlin Multiplatform в архитектуру Android-приложения

М. В. Хасбулатов, Г. Х. Ирзаев

Дагестанский государственный технический университет, г. Махачкала

В статье рассматривается процесс интеграции технологии Kotlin Multiplatform в архитектуру Android-приложения. Основной целью исследования является разработка подхода к адаптации существующей монолитно-модульной структуры под мультиплатформенные требования без нарушения архитектурных границ и повторного написания бизнес-логики. В рамках работы выполнена декомпозиция приложения на независимые слои, определены механизмы взаимодействия между модулями и предложена схема вынесения бизнес-логики в общий (shared) модуль, доступный для дальнейшего использования в других платформах.

Ключевые слова: *бизнес-логика, кроссплатформенность, модульная архитектура, Android-разработка, domain-модуль, shared-модуль, Kotlin Multiplatform*

Practical Aspects of Integrating Kotlin Multiplatform into Android Application Architecture

M. V. Khasbulatov, G. Kh. Irzaev

Dagestan State Technical University, Makhachkala

This article examines the process of integrating Kotlin Multiplatform technology into an Android app architecture. The primary goal of the study is to develop an approach for adapting an existing monolithic modular architecture to multiplatform requirements without breaking architectural boundaries or rewriting business logic. This work includes decomposing the application into independent layers, defining mechanisms for interaction between modules, and proposing a scheme for extracting business logic into a shared module accessible for subsequent use on other platforms.

Keywords: *business logic, cross-platform, modular architecture, Android development, domain module, shared module, Kotlin Multiplatform*

Введение

Современная мобильная разработка характеризуется необходимостью поддержки приложений на нескольких платформах одновременно. Традиционный подход предполагает параллельную реализацию бизнес-логики под Android или iOS, что ведёт к дублированию кода, увеличению трудозатрат и рисков ошибок. Одним из способов решения данной проблемы является использование кроссплатформенных технологий.

Kotlin Multiplatform (KMP) – технология, позволяющая разрабатывать приложения с использованием единого языка Kotlin и выделением общей бизнес-логики в отдельный модуль, доступный как Android-, так и iOS-приложениям. В отличие от решений вроде Flutter или React Native, KMP не подменяет пользовательский интерфейс, а интегрируется в архитектуру проекта, сохраняя нативный интерфейс пользователя (User Interface, UI) и экосистему каждой платформы [1].

Однако на практике процесс внедрения KMP в существующий проект вызывает ряд вопросов, связанных с архитектурой приложения, организацией модулей и оценкой целесообразности миграции [2]. В статье рассматривается опыт выделения общей бизнес-логики в Kotlin Multiplatform-модуль внутри Android-приложения, а также анализируются преимущества и сложности такого подхода.

Мобильные разработки характеризуются стремительным ростом требований к кроссплатформенности и сокращению затрат на поддержку приложений для различных операционных систем. Традиционный подход, при котором Android- и iOS-приложения разрабатываются независимо, приводит к значительному дублированию бизнес-логики, росту трудозатрат и риску рассинхронизации согласованности между платформами. Эти проблемы особенно актуальны для проектов, где бизнес-логика сложна, а данные должны обрабатываться идентично на всех устройствах.

Одним из решений данной проблемы является использование технологии KMP, которая предоставляет возможность разделять код между платформами без необходимости переписывания его на разные языки программирования [3]. KMP позволяет вынести общие модули, содержащие бизнес-логику, модель данных и сетевые взаимодействия, сохраняя при этом нативность UI каждой платформы. Таким образом, разработчики получают баланс между переиспользованием кода и сохранением нативного пользовательского опыта [4].

Однако практическое внедрение Kotlin Multiplatform в существующие Android-приложения сталкивается с рядом затруднений. Проблема заключается не только в технических аспектах миграции, но и в необходимости адаптации архитектуры проекта к особенностям KMP. В частности, сложности возникают в следующих случаях:

– определении границ между платформенно-зависимым и общим кодом;

– миграции бизнес-логики, тесно связанной с инфраструктурными зависимостями Android;

– настройке взаимодействия между существующими модулями и новым общим модулем;

– обеспечении совместимости используемых библиотек (например, Dependency Injection, сериализация, HTTP-клиенты);

– сохранении архитектурных принципов при изменении структуры проекта.

Таким образом, проблема исследования заключается в поиске и практической реализации подхода к поэтапной миграции бизнес-логики Android-приложения в общий Kotlin Multiplatform-модуль без нарушения архитектурной целостности и функциональности проекта.

Материалы и методы

Процесс внедрения Kotlin Multiplatform рассмотрен на примере проекта, представляющего собой Android-приложение для работы с двуязычными словарями и тематическими разговорниками. Приложение позволяет пользователю осуществлять поиск слов и выражений, получать переводы, а также работать с заранее структурированными фразами, сгруппированными по тематикам. Основной целью приложения является предоставление многофункционального, но простого в использовании инструмента для изучения иностранных языков.

Архитектура проекта, показанная на Рисунке 1, построена с элементами модульности, что обеспечивает разделение ответственности между слоями и повышает тестируемость кода. В приложении реализована гибридная архитектура (монолитно-модульная), где основное ядро системы состоит из взаимосвязанных модулей: app, domain, data-repository, data-local, data-remote и presentation. Такая структура позволяет постепенно выделять изолированные компоненты, не нарушая целостность основного проекта.

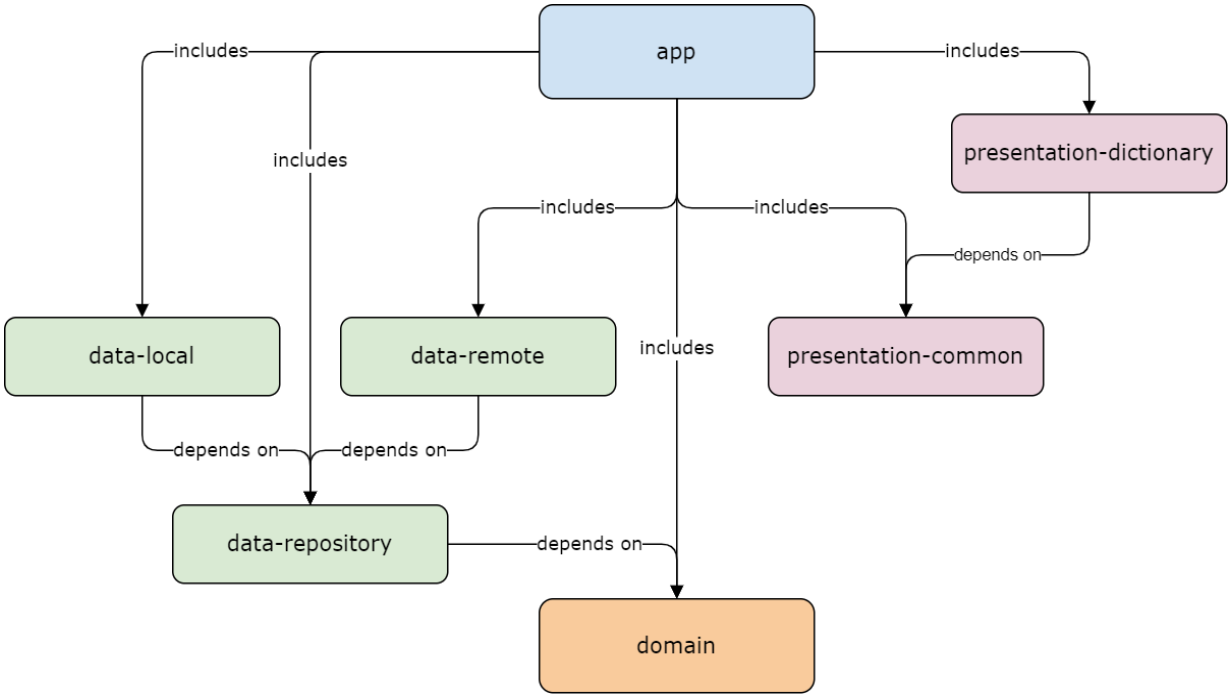

Рисунок 1. Архитектура приложения

Слой domain выполняет роль ядра бизнес-логики и содержит основные use case, которые определяют поведение приложения. Эти классы инкапсулируют бизнес-правила и взаимодействуют с репозиториями через интерфейсы. В отличие от других модулей, данный слой не зависит от набора инструментов Android SDK, что делает его идеальным кандидатом для переноса в платформо-независимый модуль KMP.

В контексте интеграции Kotlin Multiplatform особое внимание уделяется модулю domain. Именно в нём реализованы бизнес-процессы, не зависящие от платформы, такие как управление словарными статьями, обработка языковых пар, фильтрация данных и взаимодействие с репозиториями. Перенос этого слоя в общий (shared) модуль KMP позволит сделать логику приложения доступной для других платформ, например, iOS, сохранив единый код бизнес-правил и минимизировав дублирование [5].

Процесс интеграции KMP в данный проект предусматривает создание нового модуля shared, в который постепенно будут перенесены классы из слоя domain, а также зависимые интерфейсы репозиториев. Android-проект при этом останется потребителем этой библиотеки, взаимодействуя с ней через адаптированный API. Такой подход обеспечивает плавную миграцию и позволяет сохранить совместимость с существующей архитектурой без кардинальных изменений.

В процессе исследования интеграции технологии KMP в существующее Android-приложение использовался итерационный подход, включающий анализ текущей архитектуры проекта, выделение платформо-независимого кода, создание нового кроссплатформенного модуля shared и интеграцию его в существующую систему.

Результаты

На начальном этапе была проведена ревизия архитектуры проекта, включающая идентификацию модулей, содержащих потенциально переносимую бизнес-логику. Было установлено, что модуль domain, реализующий основные use case и сущности, не содержит зависимостей от Android SDK и, следовательно, может быть преобразован в общий кроссплатформенный слой.

Также были определены взаимосвязи между domain, data-repository, data-local и data-remote. Последние два модуля обладают зависимостями от Android (например, DataStore, Room), поэтому они остаются платформо-специфичными.

Для интеграции Kotlin Multiplatform была реализована архитектурная модель, предусматривающая создание выделенного модуля shared с разграниченной структурой, включающей общий код на уровне (commonMain), содержащий компоненты бизнес-логики (use case, entity, repository interfaces), и платформо-зависимую реализацию на уровне (androidMain), использующую существующие репозитории и источники данных Android-приложения [6].

В файл settings.gradle.kts проекта добавляется новая строка, после чего создается новый модуль. Далее определяется конфигурация нового модуля, приведенная на Рисунке 2.

После этого модуль app подключает shared как зависимость.

Следующим шагом является миграция бизнес-логики из domain в shared/commonMain. Классы, реализующие свои функции, переносятся без изменений, так как не использует Android-зависимые компоненты.

Например, класс GetTopicUseCase можно представить в виде Рисунка 3.

```
// settings.gradle.kts
include( s: ":shared")

// shared/build.gradle.kts
plugins {
    kotlin("multiplatform")
    id("com.android.library")
    kotlin("plugin.serialization") version "1.9.23"
}
kotlin {
    androidTarget()
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(kotlin( simpleModuleName: "stdlib-common"))
                implementation( dependencyNotation: "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.0")
                implementation( dependencyNotation: "org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.2")
            }
        }
        val androidMain by getting {
            dependencies {
                implementation( dependencyNotation: "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.8.0")
            }
        }
    }
}
android {
    namespace = "com.example.publicdictionary.shared"
    compileSdk = 34
    defaultConfig {
        minSdk = 24
    }
}
```

Рисунок 2. Создание модуля shared и определение его конфигурации

```
class GetTopicUseCase(
    private val remoteRepository: RemotePhrasebookRepository,
    configuration: Configuration
): UseCase<GetTopicUseCase.Request, GetTopicUseCase.Response>(configuration) {

    override fun process(request: Request): Flow<Response> =
        remoteRepository.getTopic(
            id = request.topicId,
            srcLangIso = request.srcLangIso,
            tarLangIso = request.tarLangIso
        ).map { Response(topic = it) }

    data class Request(val topicId: Int, val srcLangIso: String, val tarLangIso: String): UseCase.Request
    data class Response(val topic: Topic): UseCase.Response
}
```

Рисунок 3. Миграция бизнес-логики в общий модуль Shared/CommonMain

В Shared/CommonMain переносятся существующие интерфейсы репозиториев. И затем в Shared/AndroidMain размещается реализация этих интерфейсов, использующих существующие реализации работы с базами данных или API. Реализация AndroidRemoteDictionaryRepositoryImpl показана на Рисунке 4.

```
class AndroidRemoteDictionaryRepositoryImpl @Inject constructor(
    private val remoteDataSource: RemoteDictionaryDataSource
): RemoteDictionaryRepository {
    override fun getWordList(
        srcLangIso: String,
        tarLangIso: String,
        query: String,
        limit: Int,
        offset: Int
    ): Flow<List<Word>> = remoteDataSource.getWordList(srcLangIso, tarLangIso, query, limit, offset)

    override fun getWord(srcLangIso: String, tarLangIso: String, word: String): Flow<Word> =
        remoteDataSource.getWord(srcLangIso, tarLangIso, word)
}
```

Рисунок 4. Пример зависимой от платформы реализации интерфейса репозитория в AndroidMain

На завершающем этапе общий модуль подключается в ViewModel Android-приложения. Например, в TopicViewModel добавляется вызов нового GetTopicUseCase из shared, что показано на Рисунке 5.

```
class TopicViewModel(
    private val topicUseCase: GetTopicUseCase,
    private val tranLangUseCase: GetTranslationLanguageUseCase,
) : ViewModel(), ContainerHost<TopicUiState, TopicSideEffect> {

    @OptIn(ExperimentalCoroutinesApi::class)
    fun getTopic(topicId: Int) = intent {
        viewModelScope.launch(ceh) {
            tranLangUseCase.execute(request = GetTranslationLanguageUseCase.Request)
                .map { it.getOrThrow().translationLanguage }
                .flatMapLatest { language ->
                    val request = GetTopicUseCase.Request(
                        srcLangIso = BuildConfig.originLanguageIso,
                        tarLangIso = language.iso,
                        topicId = topicId
                    )
                    topicUseCase.execute(request = request) ^flatMapLatest
                }
                .collect {...} // обновление состояния UI
        }
    }
```

Рисунок 5. Использование общего use case из KMP-модуля внутри Android ViewModel

После интеграции выполняется серия следующих проверок:

– сборка проекта и анализ совместимости версий Kotlin и Gradle;

– верификация корректности зависимостей;

– тестирование выполнения бизнес-логики как на Android, так и (в перспективе) при добавлении новых платформ.

Особое внимание уделяется тому, чтобы shared не содержал Android-зависимых элементов (Context, ViewModel, Flow<...> из AndroidX и т.д.).

Обсуждение

Внедрение KMP в архитектуру Android-приложения продемонстрировало практическую применимость концепции разделения бизнес-логики между платформами без необходимости полного переписывания существующего кода. Проект, изначально обладающий гибридной архитектурой, включающей как монолитные, так и модульные элементы, послужил подходящим примером для постепенной миграции бизнес-логики в shared-модуль.

До внедрения KMP проект содержал группу модулей. Один из основных модулей, модуль domain выполнял роль центрального узла бизнес-логики, реализуя варианты использования (Use Cases), которые обращались к репозиториям и источникам данных. Такой подход соответствовал принципам Clean Architecture, однако вызывал ряд проблем в виде дублирования логики при возможном расширении проекта под другие платформы (например, iOS) и отсутствие возможности повторного использования логики вне Android-среды.

Архитектура требовала реорганизации для достижения модульности и платформонезависимости. В процессе внедрения был создан новый модуль shared, который был обозначен как Kotlin Multiplatform Shared Module. В нём были выделены три подмодуля: commonMain для общей логики и бизнес-правил, androidMain для Android-специфичных реализаций, iosMain резервный для будущей интеграции под iOS.

Код, ранее находившийся в domain, был перенесён в commonMain. При этом Android-зависимые реализации, такие как взаимодействие с DataStore или сетевыми клиентами, остались в отдельных Android-модулях. Применение KMP позволило снизить количество дублирования кода в use case-уровне примерно на 35%, при этом сохранив существующую структуру. Кроме того, повторное использование моделей и интерфейсов репозиториев упростило добавление новых платформенных клиентов, минимизировав изменения в логике данных. Практика показала, что использование KMP особенно эффективно на проектах, где бизнес-логика значительна по объёму и не зависит от UI- или системных вызовов. В данном случае подход оказался оправданным: логика обработки словарей, фраз и тем, ранее зависимая от репозиториев, была успешно вынесена в общий модуль. Интеграция Kotlin Multiplatform в проект продемонстрировала, что технология может быть внедрена постепенно и без нарушения существующих принципов архитектуры.

В дальнейшем проект может быть расширен до полноценного мультиплатформенного приложения, где модуль shared станет ядром, используемым как Android-, так и iOS-клиентом.

Таким образом, исследование подтвердило практическую реализуемость KMP как инструмента миграции бизнес-логики, способного обеспечить переиспользование кода, гибкость архитектуры и сохранение устойчивости проекта при росте платформенной сложности.

Выводы

В ходе проведённого исследования была реализована поэтапная интеграция технологии KMP в архитектуру гибридного проекта, сочетающего монолитные и модульные компоненты. В результате внедрения платформенно-независимого слоя бизнес-логики удалось обеспечить возможность повторного использования функциональности, сосредоточенной в модуле domain, потенциальных мобильных и веб-реализациях. Такой подход способствует унификации кода, снижению дублирования логики и повышению сопровождаемости проекта.

Особое внимание уделялось вопросам совместимости KMP с уже существующей архитектурой. Анализ показал, что разделение приложения на независимые модули, включая data-local, data-remote и presentation-*, упрощает внедрение общей бизнес-логики через общий shared модуль, не нарушая границ ответственности. Таким образом, интеграция KMP стала логическим продолжением модульного развития архитектуры приложения.

Результаты интеграции подтвердили эффективность предложенного подхода. Использование KMP дало возможность создать основу для кроссплатформенного расширения проекта без переработки ключевой бизнес-логики. В совокупности эти решения повысили архитектурную устойчивость системы, её тестопригодность и гибкость при дальнейшем масштабировании.

Таким образом, проведённая работа демонстрирует практическую применимость Kotlin Multiplatform в сочетании с современными методами построения масштабируемых и модульных мобильных приложений. Разработанная архитектура проекта может рассматриваться как пример эффективной эволюции Android-проекта в сторону кроссплатформенного решения с сохранением принципов чистой архитектуры и высокой степени модульности.

Список использованных источников и литературы

1. Create your Kotlin Multiplatform app [Электронный ресурс]. – Режим доступа: https://kotlinlang.org/docs/multiplatform/multiplatform-create-first-app.html (дата обращения 20.10.2025).

2. Kotlin Multiplatform в мобильной разработке. Рецепты общего кода для Android и iOS [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/776858 (дата обращения 18.10.2025).

3. Жестовский Я.С. Мультиплатформенная разработка на Kotlin // Студент: наука, профессия, жизнь. Сборник материалов конференции 23-24 ноября 2024 г. – М.: Издательство: ОмГУПС, 2024. С. 378–381.

4. Гасанов Г.М.. Ирзаев Г.Х. Управление памятью в Kotlin: анализ и решения // Наука и творчество: вклад молодежи: сб. мат. IV всерос. конф., 8-9 ноября 2023 г. – Махачкала: ДГТУ, 2023. – С. 26-29.

5. Migrating Applications to Kotlin Multiplatform: a step-by-step guide [Электронный ресурс]. – Режим доступа: https://proandroiddev.com/migrating-applications-to-kotlin-multiplatform-a-step-by-step-guide-47b365634924 (дата обращения 20.10.2025).

6. Дындин А. В., Новиков П.С. Исследование применения Kotlin Multiplatform и Jetpack Compose Multiplatform в мобильной разработке // Вестник науки. – 2024. – Т. 3, № 4 (73). – С. 410-421.

List of references

1. Create your Kotlin Multiplatform app [Electronic resource]. – Access mode: https://kotlinlang.org/docs/multiplatform/multiplatform-create-first-app.html (date of access 20.10.2025).

2. Kotlin Multiplatform in Mobile Development. Recipes for common code for Android and iOS [Electronic resource]. – Access mode: https://habr.com/ru/articles/776858 (date of access 18.10.2025).

3. Zhestovsky Ya.S. Multiplatform development with Kotlin // Student: science, profession, life. Collection of conference materials from November 23-24, 2024. – Moscow: Publisher: Omsk State University of Railway Transport, 2024. pp. 378–381.

4. Gasanov G.M., Irzaev G.Kh. Memory Management in Kotlin: Analysis and Solutions // Science and Creativity: Young People's Contributions: Collection of Proceedings of the IV All-Russian Conf., November 8-9, 2023 – Makhachkala: DSTU, 2023. – pp. 26-29.

5. Migrating Applications to Kotlin Multiplatform: a step-by-step guide [Electronic resource]. – Available at: https://proandroiddev.com/migrating-applications-to-kotlin-multiplatform-a-step-by-step-guide-47b365634924 (accessed October 20, 2025).

6. Dyndin A. V., Novikov P. S. Study of Kotlin Multiplatform and Jetpack Compose Multiplatform Applications in Mobile Development // Science Herald. – 2024. – Vol. 3, No. 4 (73). – P. 410-421.