УДК 004 ***Дындин А.В., Новиков П.С.***

Дындин А.В.

студент

Московский политехнический университет

(г. Москва, Россия)

Новиков П.С.

к.т.н., доцент

Московский политехнический университет

(г. Москва, Россия)

ИССЛЕДОВАНИЕ ПРИМЕНЕНИЯ KOTLIN MULTIPLATFORM И JETPACK COMPOSE MULTIPLATFORM В МОБИЛЬНОЙ РАЗРАБОТКЕ

Аннотация: *в данной статье проводится исследование применения Kotlin Multiplatform и Jetpack Compose Multiplatform в мобильной разработке. Рассматривается возможность использования единой кодовой базы для создания кроссплатформенных приложений, что позволяет разработчикам унифицировать процессы разработки и поддержки программного обеспечения на различных платформах, включая Android и iOS. Статья описывает основные концепции, преимущества и технические аспекты Kotlin Multiplatform и Compose Multiplatform, предоставляя анализ их функциональности через механизмы такие как "expect/actual" и Dependency Injection. Также приводятся примеры кода, что подчеркивает упрощение процесса создания UI. В заключение описываются текущие тренды и потенциальное будущее мультиплатформенных технологий в индустрии программного обеспечения.*

Ключевые слова: *кроссплатформенная разработка, мультиплатформенные приложения, производительность приложений, унификация кодовой базы.*

Быстрая эволюция мобильных технологий и возрастающий спрос на универсальные программные решения породили необходимость в разработке

фреймворков и платформ, которые могли бы одновременно обслуживать несколько систем. Kotlin Multiplatform (KMP) и Jetpack Compose Multiplatform выступают ключевыми технологиями в этом направлении, предлагая уникальную возможность разработки кросс-платформенных приложений с использованием единой кодовой базы. Это не только сокращает время и ресурсы, затрачиваемые на разработку и поддержку приложений, но и улучшает консистенцию и качество продуктов.

Цель исследования – анализ применения Kotlin Multiplatform и Jetpack Compose Multiplatform для мобильной разработки. Особое внимание уделяется их способности упростить создание мультиплатформенных приложений, обеспечивая высокую производительность и оптимальное взаимодействие с пользователем на различных устройствах. Исследование направлено на выявление основных преимуществ, технических аспектов и потенциальных ограничений данных технологий, а также их влияние на современные подходы в разработке программного обеспечения. Это исследование предоставит ценные инсайты разработчикам, стремящимся оптимизировать процессы создания программных продуктов и улучшить их качество с помощью мультиплатформенных технологий.

Актуальность.

В последние годы мультиплатформенная разработка набирает популярность, что способствовано стремлению к экономии ресурсов и ускорению процесса вывода продуктов на рынок. Значительный вклад в эту область вносят компании Google и JetBrains через разработку фреймворков, таких как Flutter, Jetpack Compose и Kotlin Multiplatform, которые упрощают создание приложений, используя единую кодовую базу для всех платформ.

Google с Jetpack Compose Multiplatform и JetBrains с Kotlin Multiplatform предоставляют инструменты для унификации кода и интерфейсов, сокращая время разработки и повышая качество продуктов. Jetpack Compose облегчает разработку UI, предлагая декларативный способ описания интерфейсов, который

компилируется в нативный код. Kotlin Multiplatform позволяет использовать общую бизнес-логику на различных платформах, сокращая количество ошибок и дублирования кода.

Такие технологии значительно влияют на индустрию, делая мультиплатформенную разработку не только экономически выгодной, но и технологически продвинутой, что подтверждает важность их дальнейшего изучения и развития.

Kotlin Multiplatform.

Kotlin Multiplatform (KMP) представляет собой мощный инструмент для разработки мультиплатформенных приложений, позволяющий разработчикам использовать один и тот же код на различных платформах, включая Android, iOS, Web и Desktop. Основная идея KMP заключается в том, что он позволяет писать общую логику на Kotlin, которая компилируется в нативный код для каждой платформы, обеспечивая высокую производительность и интеграцию с платформо-специфическими функциями, что является значительным преимуществом перед другими кроссплатформенными инструментами, которые могут сталкиваться с ограничениями производительности и доступности функций.

Использование KMP значительно упрощает процесс портирования кода с Android на другие платформы. Благодаря возможности разделять общую логику и платформо-специфический код, разработчики могут легко адаптировать свои приложения под нужные платформы без необходимости переписывания кода с нуля. Это облегчает поддержку и развитие приложений, сокращает время на разработку и тестирование.

В проектах используются такие модули как:

- commonMain – управляет бизнес-логикой и UI компонентами, которые могут быть переиспользованы на любой платформе
- androidMain, iosMain и подобные – платформо-специфичные модули, которые адаптируют логику и интерфейсы к особенностям и требованиям

конкретных платформ, используя нативное API и библиотеки для оптимальной производительности и интеграции.

Один из ключевых моментов при использовании KMP – это возможность абстрагирования платформо-специфического кода через механизмы expect и actual. Эти механизмы позволяют разработчикам объявлять ожидаемые (expected) интерфейсы или функции, которые должны быть реализованы для каждой целевой платформы (actual). Использование Dependency Injection (DI) может помочь управлять этими зависимостями эффективно, адаптируя код под конкретную платформу без изменения бизнес-логики.

Dependency Injection (DI) — это программный паттерн проектирования, используемый для управления зависимостями между компонентами в приложениях. Основная идея DI заключается в том, чтобы уменьшить связность между компонентами приложения, что облегчает управление зависимостями и усиливает модульность системы.

Основные компоненты

- expect declaration - определяет общий интерфейс или функцию, которая будет специализирована для каждой платформы.
- actual implementation - специфичные реализации для каждой платформы, соответствующие объявленным expect.

Сначала определим expect класс, который объявляет функцию загрузки данных.

```
// CommonMain.kt
expect class NetworkClient() {
    fun get(url: String): String
}
```

Рис. 1. Объявление класса запроса в сеть на KMP.

Затем предоставим платформо-специфические реализации этой функции для Android и iOS.

```
// AndroidMain.kt
actual class NetworkClient actual constructor() {
    actual fun get(url: String): String {
        // Использование OkHttp для отправки HTTP запросов на Android
        val client = OkHttpClient()
        val request = Request.Builder()
            .url(url)
            .build()
        val response = client.newCall(request).execute()
        return response.body?.string() ?: ""
    }
}

// iOSMain.kt
actual class NetworkClient actual constructor() {
    actual fun get(url: String): String {
        // Использование NSURLSession для отправки HTTP запросов на iOS
        val url = NSURL.URLWithString(url)!!
        val request = NSMutableURLRequest.requestWithURL(url)
        var responseString = ""
        NSURLSession.sharedSession()
            .dataTaskWithRequest(request) { data, resp, err ->
                responseString = NSString
                    .create(data!!, NSStringEncoding.UTF8StringEncoding) as String
        }.resume()
        return responseString
    }
}
```

Рис. 2. Реализация сетевого запроса на платформах Android и iOS с использованием KMP.

Теперь интегрируем эти реализации с системой DI для управления зависимостями, используя Kodein DI.

```
// CommonMain.kt
fun DI.MainBuilder.bindNetworkClient() {
    bind<NetworkClient>() with singleton { NetworkClient() }
}

// App.kt
val di = DI {
    bindNetworkClient()  // Включение NetworkClient в граф зависимостей
}

// Использование
val networkClient: NetworkClient by di.instance()
val data = networkClient.get("https://api.example.com/data")
```

Рис. 3. Использование Kodein DI для создания сетевого клиента.

В этом примере NetworkClient определяется как expect класс с методом get, что позволяет абстрагировать детали реализации сетевых запросов на уровне платформы. Реализации для Android и iOS указываются как actual, с использованием платформо-специфических API.

DI обеспечивает возможность внедрения зависимостей в компоненты приложения во время выполнения, что позволяет легко подменять реализации в зависимости от платформы, что особенно актуально при использовании KMP, так как разные платформы могут требовать разных реализаций для одних и тех же задач, например, обработки HTTP-запросов или работы с базой данных.

Этот подход позволяет сохранять код чистым и модульным, а также упрощает тестирование и поддержку приложения, разделяя бизнес-логику и платформо-специфическую реализацию.

Jetpack Compose Multiplatform.

Compose Multiplatform, разработанный как расширение популярного фреймворка Jetpack Compose от Google, предоставляет разработчикам инструменты для создания декларативных UI, которые могут компилироваться и работать на нескольких платформах, включая Android, iOS, Web и Desktop. Это

решение позволяет использовать единую кодовую базу для всех платформ, значительно сокращая объем работы по поддержке различных систем.

В сравнении с другими UI фреймворками, такими как Flutter или React Native, Compose Multiplatform выделяется тем, что позволяет разработчикам использовать полный стек технологий Kotlin, обеспечивая бесшовную интеграцию с существующими проектами Kotlin и Jetpack Compose на Android.

Compose Multiplatform значительно упрощает процесс создания кроссплатформенных пользовательских интерфейсов благодаря единому языку программирования (Kotlin) и единообразию API по всем платформам. Разработчики могут создавать компоненты интерфейса один раз и использовать их на любой платформе без необходимости адаптации под спецификации каждой из них.

Для демонстрации использования Compose Multiplatform в сочетании с реализованным сетевым клиентом на Kotlin Multiplatform создадим Composable функцию, которая будет использоваться для отображения данных на экране.

Помеченные @Composable аннотацией функции могут вызывать другие Composable функции, образуя иерархическое дерево UI компонентов, что улучшает модульность и повторное использование кода. Этот подход упрощает разработку интерфейса, делая его более интуитивно понятным и легко адаптируемым под динамичные данные, а также позволяет интерфейсу автоматически обновляться при изменениях состояний, позволяя приложению работать оптимальнее и эффективнее.

```kotlin
// commonMain/kotlin/MainScreen.kt

@Composable
fun MainScreen(di: DI) {
    var text by remember { mutableStateOf("Loading...") }

    // Получаем сетевой клиент из DI
    val networkClient: NetworkClient by di.instance()

    // Делаем сетевой запрос
    LaunchedEffect(true) {
        text = networkClient.get("https://api.example.com/data")
    }

    // Выводим результат сетевого запроса в виде текста
    Text(text = text)
}
```

Рис. 4. Реализация пользовательского интерфейса с использованием Jetpack Compose.

На Android, MainActivity должна использовать функцию MainScreen() для отображения пользовательского интерфейса.

```kotlin
// androidMain/java/com/example/myapplication/MainActivity.kt
class MainActivity : AppCompatActivity() {
    private val di = DI {
        import(commonDiModule)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainScreen(di = di)
        }
    }
}
```

Рис. 5. Отображение пользовательского интерфейса на Android.

Для iOS, подход будет аналогичен, используя SwiftUI или другой метод для интеграции Kotlin Multiplatform и Compose Multiplatform.

```swift
// iosMain/swift/ViewController.swift
class ViewController: UIViewController {
    private var di = DI {
        import(commonDiModule)
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        setupComposeView()
    }

    func setupComposeView() {
        let rootView = MainScreenKt.MainScreen(di: di)
        let hostingController = UIHostingController(rootView: rootView)
        addChild(hostingController)
        view.addSubview(hostingController.view)
        setupConstraints(for: hostingController.view)
        hostingController.didMove(toParent: self)
    }
}
```

Рис. 6. Отображение пользовательского интерфейса на iOS.

В этом примере был показан процесс интеграции мультиплатформенного сетевого запроса с пользовательским интерфейсом, созданным с помощью Compose Multiplatform. Используя Kodein DI для управления зависимостями и NetworkClient для выполнения HTTP-запросов, данная реализация демонстрирует гибкость и мощь KMP в создании кроссплатформенных приложений с реактивными пользовательскими интерфейсами.

Скорость разработки.

При создании концепта приложения разработчики старались максимально эффективно использовать KMP и Jetpack Compose Multiplatform для минимизации количества платформозависимого кода, после чего вывели данную диаграмму, показывающую процент затраченных усилий при использовании нативной разработки под Android и iOS в сравнении с применением вышеописанных технологий.

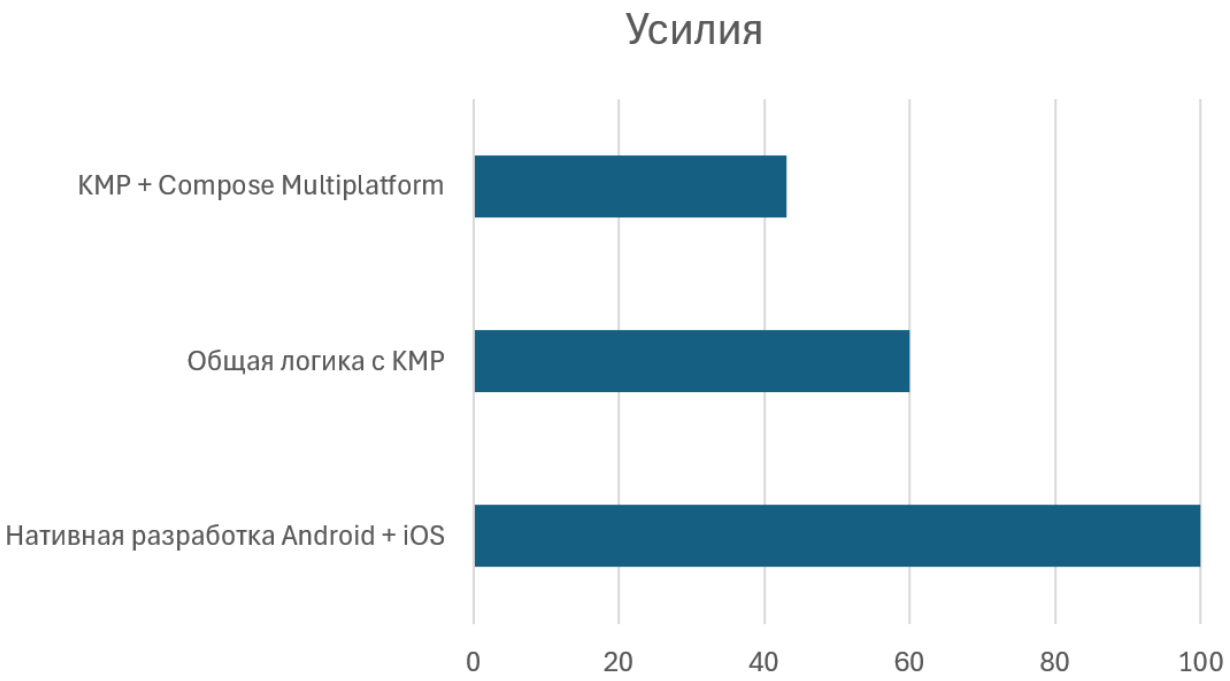

Рис. 7. Уровень усилий, необходимых в различных сценариях, с использованием KMP, Jetpack Compose и без них.

Выводы из данного раздела подчеркивают, что KMP с использованием Jetpack Compose Multiplatform предлагает разработчикам гибкое и мощное средство для создания надежных мультиплатформенных приложений, существенно упрощая процесс разработки и обеспечивая высокое качество конечного продукта благодаря возможности использования нативного кода и глубокой интеграции с платформами.

Заключение.

В ходе исследования было подтверждено, что Kotlin Multiplatform и Compose Multiplatform играют важную роль в современной разработке программного обеспечения. Эти технологии предоставляют разработчикам мощные инструменты для создания мультиплатформенных приложений, позволяя использовать единую кодовую базу для Android, iOS, Web и других платформ. Особенно выделяется возможность Compose Multiplatform упрощать разработку пользовательских интерфейсов, делая их визуально согласованными и функционально идентичными на различных устройствах.

Текущие тренды показывают, что рынок все более и более склоняется к использованию мультиплатформенных решений, поскольку они обеспечивают значительное сокращение времени и ресурсов, необходимых для разработки и поддержки приложений. Kotlin Multiplatform и Compose Multiplatform выступают в этом контексте не только как инструменты для повышения эффективности, но и как средства для достижения большей гибкости в процессе разработки.

Потенциальное влияние на индустрию разработки программного обеспечения огромно. Улучшение производительности и удобства разработки приложений, которые легко масштабируются и адаптируются к различным платформам и устройствам, может радикально изменить подходы к проектированию и разработке в IT-секторе. Со временем это приведет к более быстрому внедрению новых приложений на рынок, улучшению качества продуктов и, как следствие, к повышению удовлетворенности пользователей.

В заключение, Kotlin Multiplatform и Compose Multiplatform представляют собой ключевые элементы современной разработки приложений, способствующие эволюции мультиплатформенных решений. Их роль и значение будут только расти по мере того, как всё больше компаний и разработчиков будут стремиться к оптимизации своих рабочих процессов и созданию высококачественных, универсальных приложений.

СПИСОК ЛИТЕРАТУРЫ:

1. Kotlin Multiplatform в мобильной разработке. Рецепты общего кода для Android и iOS [Электронный ресурс]. URL: https://habr.com/ru/articles/776858/ (дата обращения: 20.04.24);
2. Get started with Kotlin Multiplatform [Электронный ресурс]. URL: https://www.jetbrains.com/help/kotlin-multiplatform-dev/get-started.html (дата обращения: 20.04.24);

3. Compose Multiplatform [Электронный ресурс]. URL: https://www.jetbrains.com/lp/compose-multiplatform/ (дата обращения: 20.04.24);
4. Implementing a 2-month Large-scale Banking POC with Kotlin Multiplatform [Электронный ресурс]. URL: https://blog.apter.tech/implementing-a-2-month-large-scale-banking-poc-with-kotlin-multiplatform-mobile-12ab887afb (дата обращения: 20.04.24)

Dyndin A.V., Novikov P.S.

Dyndin A.V.
Moscow Polytechnic University
(Moscow, Russia)

Novikov P.S.
Moscow Polytechnic University
(Moscow, Russia)

STUDY OF THE APPLICATION OF KOTLIN MULTIPLATFORM AND JETPACK COMPOSE MULTIPLATFORM IN MOBILE DEVELOPMENT

Abstract*: this article investigates the application of Kotlin Multiplatform and Jetpack Compose Multiplatform in mobile development. It explores the possibility of using a unified codebase to create cross-platform applications, which allows developers to standardize the processes of development and support of software across various platforms, including Android and iOS. The article describes the fundamental concepts, benefits, and technical aspects of Kotlin Multiplatform and Compose Multiplatform, providing an analysis of their functionalities through mechanisms such as "expect/actual" and Dependency Injection. It also includes code examples that highlight the simplification of the UI creation process. In conclusion, the article discusses current trends and the potential future of multiplatform technologies in the software industry.*

Keywords*: Kotlin Multiplatform, Jetpack Compose, Crossplatform Development, Multiplatform Applications, Application Performance, Codebase Unification.*