

Кроссплатформенная разработка КМР/СМР

[Задания на практику](#)



Введение

Наша задача - создать мультиплатформенное приложение “GoElephants”.

Суть приложение такова:

- на первом экране будет отображаться случайный слон с его фотографией и описанием по нажатию кнопки,
- на втором экране - нужно пройти тест, чтобы определить, какой ты слон.

Стек: Kotlin Multiplatform и Compose Multiplatform

Данные о слонах будем получать через API (ознакомьтесь с ним): <https://elephants.caravanlabs.ru/docs>



Рекомендуем обращаться к sample мультиплатформенного приложения:

<https://github.com/Radch-enko/EffectiveCelestia>



Архитектура проекта

Рассмотрим из чего состоит репозиторий

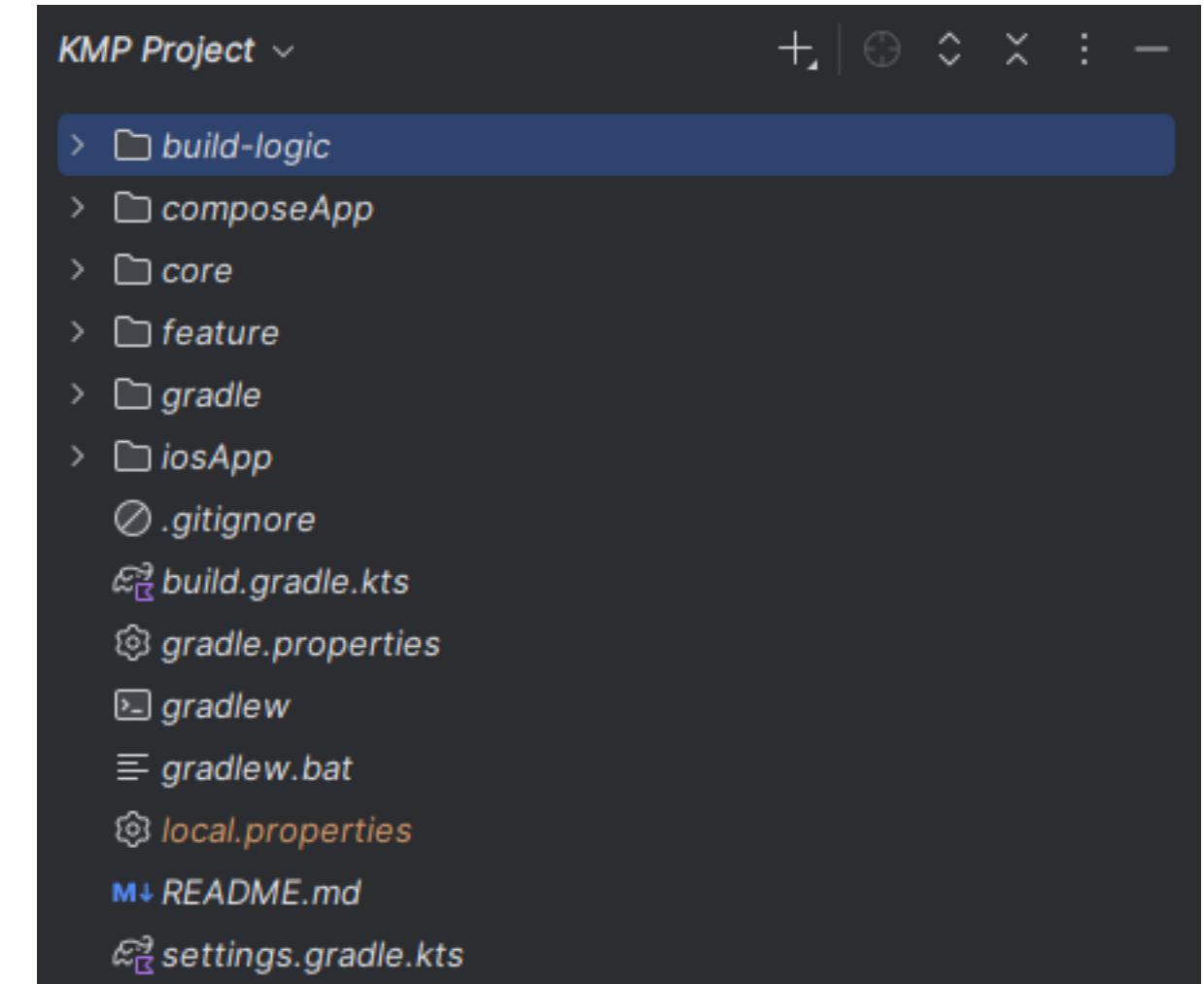
/build-logic - это служебный Gradle-модуль для централизованного управления логикой сборки проекта. Вместо того чтобы копировать одинаковые настройки в build.gradle.kts каждого модуля, эта логика выносится в build-logic и переиспользуется

/composeApp - главный модуль-интегратор. Это точка входа, которая объединяет все feature-модули и core-компоненты в единое Compose Multiplatform-приложение. Его основная задача - навигация и компоновка UI.

- commonMain - содержит общую логику UI: корневой компонент App() и навигацию
- androidMain, iosMain, jvmMain - платформенно-специфичная реализация (например, работа с системными API или навигацией)

/iosApp - нативный iOS-проект, являющийся точкой входа для приложения на iOS. Он может содержать SwiftUI-код или просто оборачивать общий UI из composeApp

/core и /feature - эти модули создаются для вынесения общих компонентов (core) и изолирования функциональных возможностей (feature)

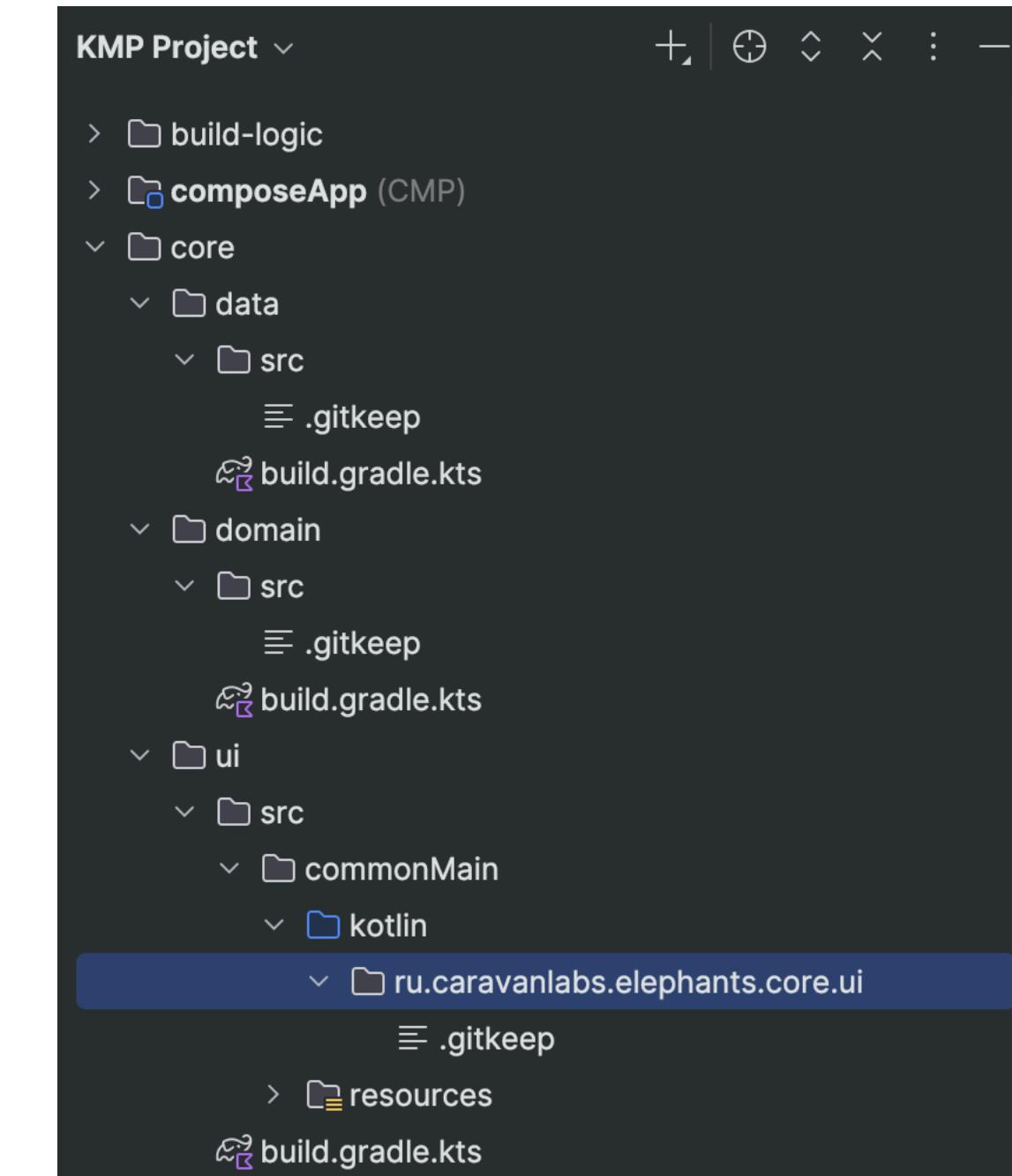


1. Core-модуль

Модуль `core` служит фундаментом приложения, объединяя код, который повторно используется во всех функциональных модулях. Для обеспечения чёткой структуры и разделения ответственности он организован по трём архитектурным слоям:

- Data — источники данных и репозитории
 - Domain — бизнес-модели и сценарии (Use Cases)
 - UI — базовые компоненты интерфейса и темы

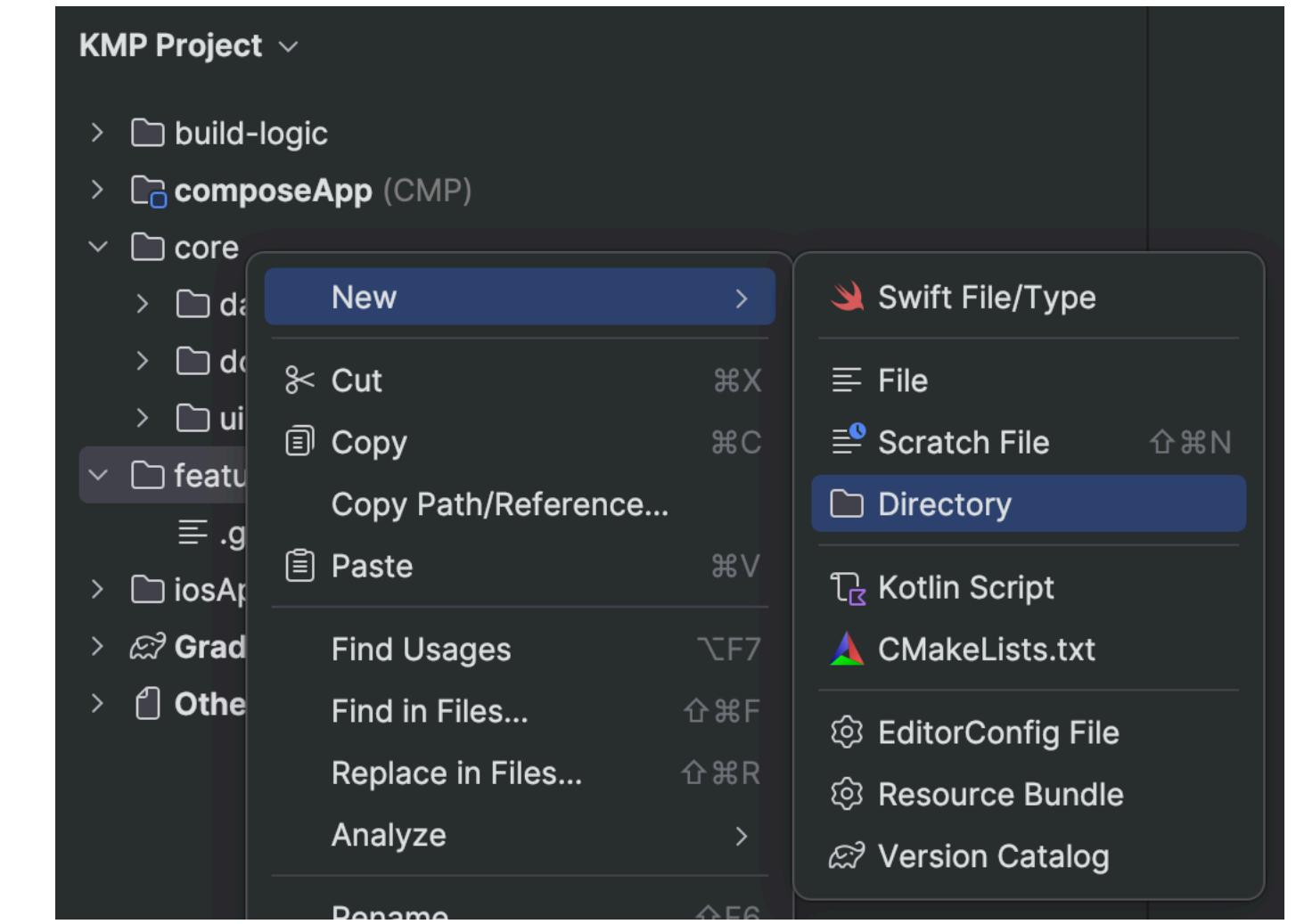
Задание: реализуйте архитектурные слои Data и Domain по аналогии с уже созданным слоем UI (core/ui). Создайте соответствующие пакеты.



2.1 Создание feature-модуля

Для реализации первого экрана приложения необходимо создать изолированный feature-модуль (например, `randomelephant`). Этот модуль будет отвечать за всю логику получения и отображения случайного слона: от запроса к API и обработки данных до пользовательского интерфейса с фотографией, описанием и кнопкой обновления.

Задание: создайте новый модуль



2.2 Настройка feature-модуля

После создания модуля необходимо настроить его внутреннюю структуру и подключить к проекту

Шаги настройки:

1. Создать базовую структуру - в модуле создать директорию src и файл build.gradle.kts
2. Организовать исходный код - внутри src создать стандартные для KMP source sets (commonMain, androidMain и iosMain). Весь общий код (модели, логика, Compose UI) будет находиться в commonMain
3. Настроить зависимости - в build.gradle.kts прописать необходимые зависимости (добавить плагин, создать sourceSets для зависимостей)
4. Интегрировать модуль в проект - добавить новый модуль в список include(...) корневого settings.gradle.kts и добавить его как зависимость в composeApp/build.gradle.kts, чтобы общий UI-модуль мог его использовать

На следующем слайде представлена структура готового варианта

2.3 Настройка feature-модуля

KMP Project

+ | ⊕ ◇ × : -

build.gradle.kts (:feature:randomizer) settings.gradle.kts (GoElephants)

```

You can use the Project Structure dialog to view... Open (⌘;) Hide notification
You can use the Project Struct... Open (⌘;) Hide notification

```

```

1  plugins {
2     id("ru.caravanlabs.elephants.convention.mobilefeature")
3 }
4
5 kotlin {
6     sourceSets {
7         commonMain.dependencies {
8             ...
9         }
10        androidMain.dependencies {
11            ...
12        }
13        iosMain.dependencies {
14            ...
15        }
16    }
17 }
18
19 compose.resources {
20     publicResClass = true
21     generateResClass = auto
22 }
23
24
25
26
27
28
29
30
31
32 include( ...projectPaths = ":composeApp")
33 include( ...projectPaths = ":core:data")
34 include( ...projectPaths = ":core:domain")
35 include( ...projectPaths = ":core:ui")
36 include( ...projectPaths = ":feature:randomizer")

```

3. Реализация слоёв в feature-модуле

Внутри модуля commonMain организуйте код согласно чистой архитектуре, создав три слоя:

1. Domain – бизнес-модели (data class) и интерфейсы репозиториев
2. Data – реализация репозиториев и работа с источником данных
3. Presentation (UI) – ViewModel (для управления состоянием) и экран на Compose

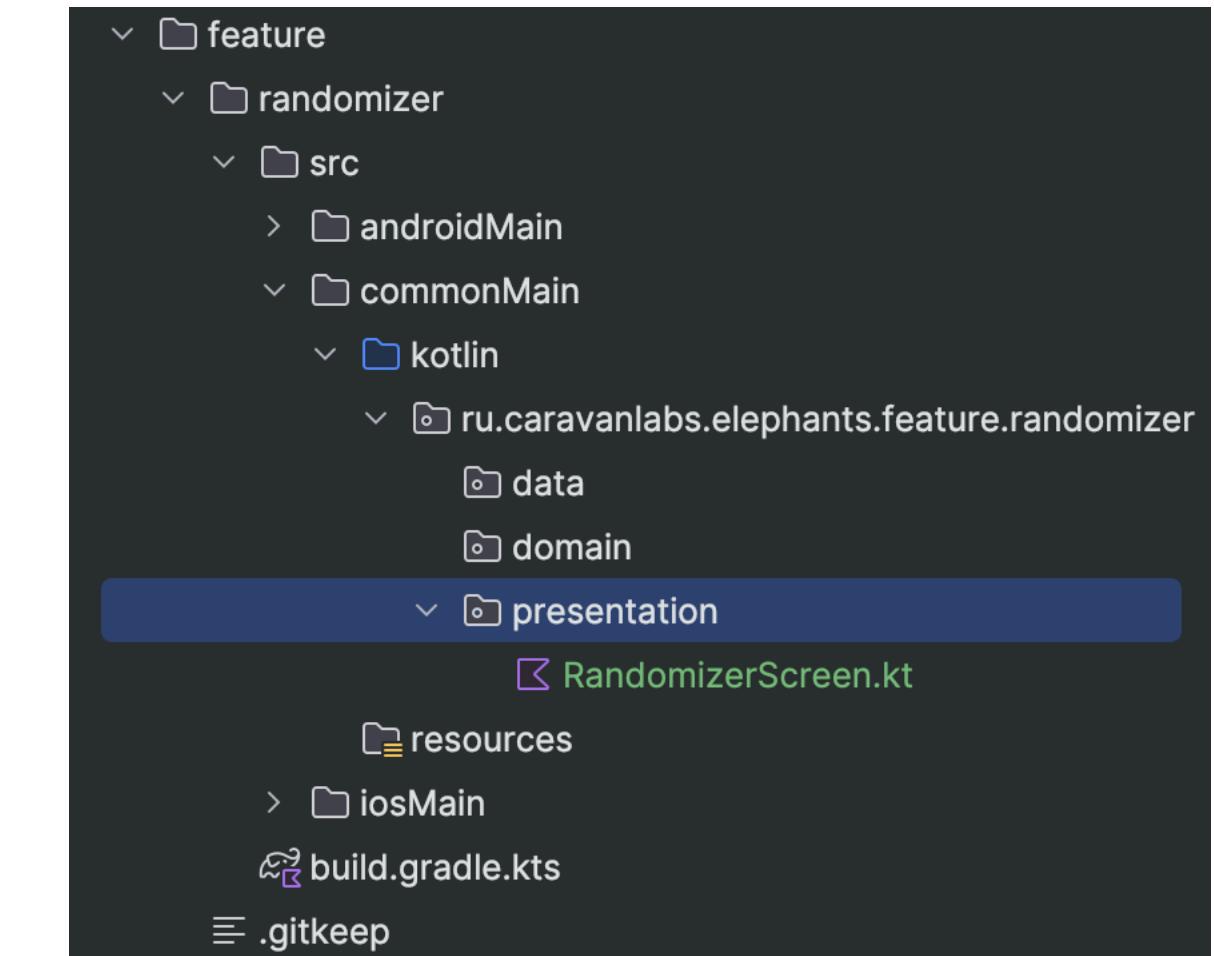
- Верстка экрана и загрузка изображений

Начните разработку в слое Presentation. Для отображения фотографии слона по ссылке используйте AsyncImage из библиотеки Coil. Это стандартное решение для эффективной и удобной загрузки изображений в Compose.

Полезные материалы:

- [Официальная документация по загрузке изображений в Compose](#)
- [Репозиторий и руководство по Coil](#)

(как добавлять библиотеки – смотри на следующем слайде)



4. Подключение зависимостей

Чтобы использовать библиотеку в feature-модуле, добавьте её в build.gradle.kts

Следуйте официальной документации, но учитывайте специфику KMP:

1. Откройте build.gradle.kts вашего feature-модуля.
2. Найдите нужный sourceSet (например, commonMain для кода, общего на всех платформах).
3. Вставьте зависимость в блок dependencies { ... }. Современные IDE часто подсказывают перенести её в централизованный каталог версий (например, libs.versions.toml) — рекомендуется следовать этой подсказке для удобства управления.

Ключевое замечание для KMP:

Всегда проверяйте, что библиотека поддерживает мультиплатформенность. Например, для загрузки изображений в KMP через Coil следует использовать coil-network-ktor3, а не coil-network-okhttp, так как Ktor является мультиплатформенным HTTP-клиентом

```

KMP Project ▾
build.gradle.kts
> core
< feature
  < randomizer
    < src
      > androidMain
      < commonMain
        < kotlin
          < ru.caravanlabs.elephants.feature.randomizer
            < data
            < domain
            > presentation
            < composeResources
        > iosMain
    < build.gradle.kts
build.gradle.kts (:feature:randomizer) ×
Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. Sync Now Ignore
1 plugins {
2     id("ru.caravanlabs.elephants.convention.mobilefeature")
3 }
4
5 kotlin {
6     sourceSets {
7         commonMain.dependencies {
8             implementation(dependencyNotation = "io.coil-kt.coil3:coil-compose:3.3.0")
9             implementation(dependencyNotation = "io.coil-kt.coil3:coil-network-okhttp:3.3.0")
10        }
11    }
12    androidMain.dependencies {
13    }
14 }
15

```

build.gradle.kts (:feature:randomizer) ×

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. Sync Now Ignore

plugins {
 id("ru.caravanlabs.elephants.convention.mobilefeature")
}

kotlin {
 sourceSets {
 commonMain.dependencies {
 implementation(dependencyNotation = "io.coil-kt.coil3:coil-compose:3.3.0")
 implementation(dependencyNotation = "io.coil-kt.coil3:coil-network-okhttp:3.3.0")
 }
 }
 androidMain.dependencies {
 }
}

Use version catalog instead Toggle info (⌘F1)
Replace with new library catalog declaration, reusing version variable coilCompose

5. Создание State и ViewModel для экрана

Помимо описания компоновки UI в RandomizerScreen, необходимо реализовать управление состоянием экрана

Задание:

- Создайте состояние State, которое будет содержать все данные для отображения
- Создайте класс ViewModel, который хранит состояние, т.е. содержит изменяемый (MutableStateFlow) и публичный неизменяемый (StateFlow) поток состояния (он же State)

The screenshot shows a KMP Project structure in a code editor. The project tree on the left includes build-logic, composeApp, core, feature (with randomizer), and presentation (containing RandomizerScreen.kt). The RandomizerState file (highlighted in blue) contains the following code:

```
1 package ru.caravanlabs.elephants.feature.randomizer
2
3 data class RandomizerState(
4     val isLoading: Boolean
5 )
6
7
```

The RandomizerViewModel file contains the following code:

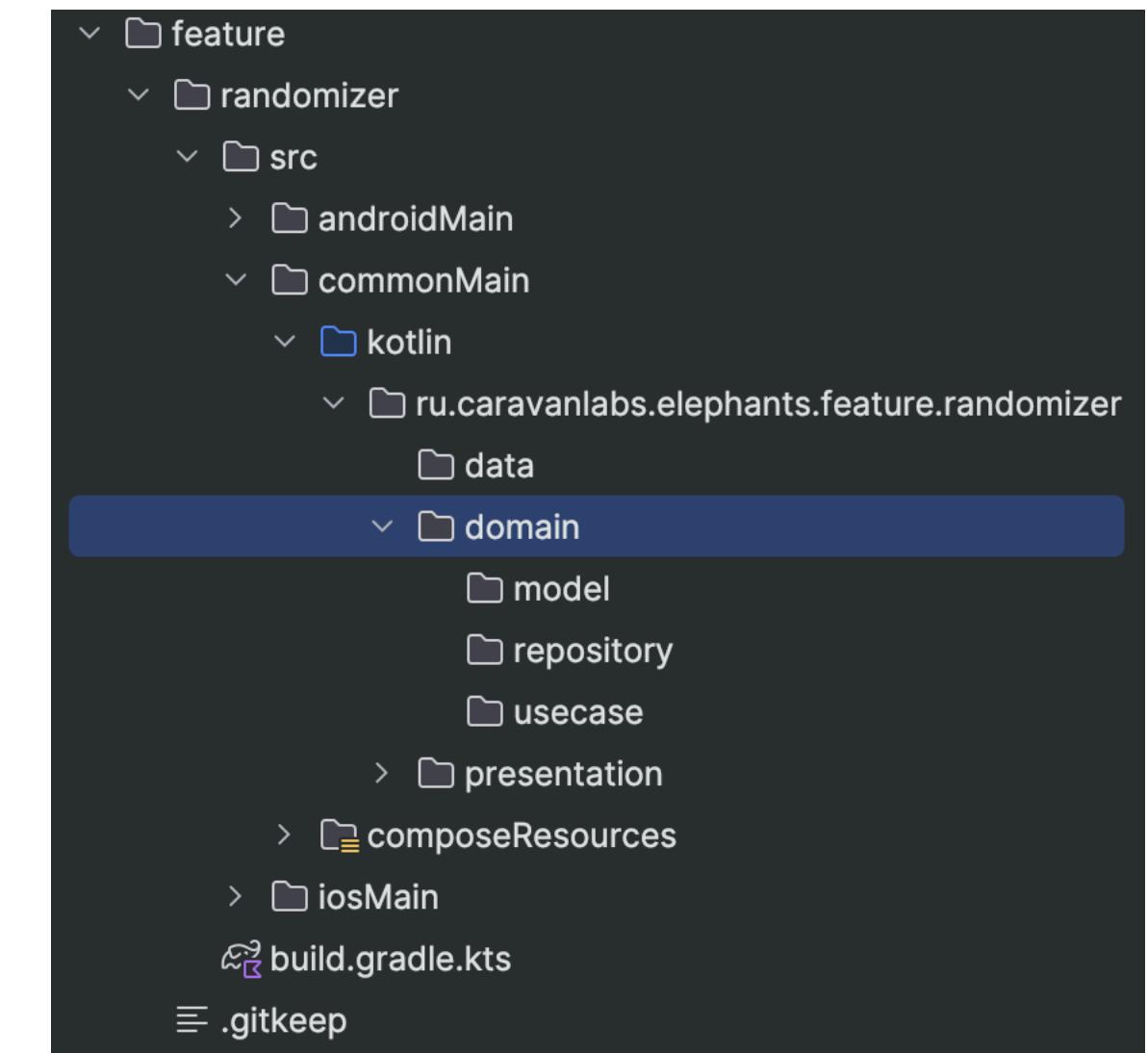
```
1 package ru.caravanlabs.elephants.feature.randomizer
2
3 import androidx.lifecycle.ViewModel
4 import kotlinx.coroutines.flow.MutableStateFlow
5 import kotlinx.coroutines.flow.asStateFlow
6
7 class RandomizerViewModel(): ViewModel() {
8
9     private val mutableState = MutableStateFlow(
10         value = RandomizerState(isLoading = true)
11     )
12
13     val state = mutableState.asStateFlow()
14 }
```

6. Domain-слой

Данный слой инкапсулирует ядро приложения, не завися от внешних слоёв, что обеспечивает независимость бизнес-правил.

Компоненты:

1. Модели (Entities) - data class для описания сущностей (например, Elephant).
2. Интерфейсы Repository - определяют контракты доступа к данным ("что нужно"), без деталей реализации ("как")
3. UseCases - классы с единой ответственностью, которые реализуют конкретные бизнес-сценарии. Каждый UseCase описывается интерфейсом, реализация помещается в пакет `impl`



Задание: создайте в слое domain

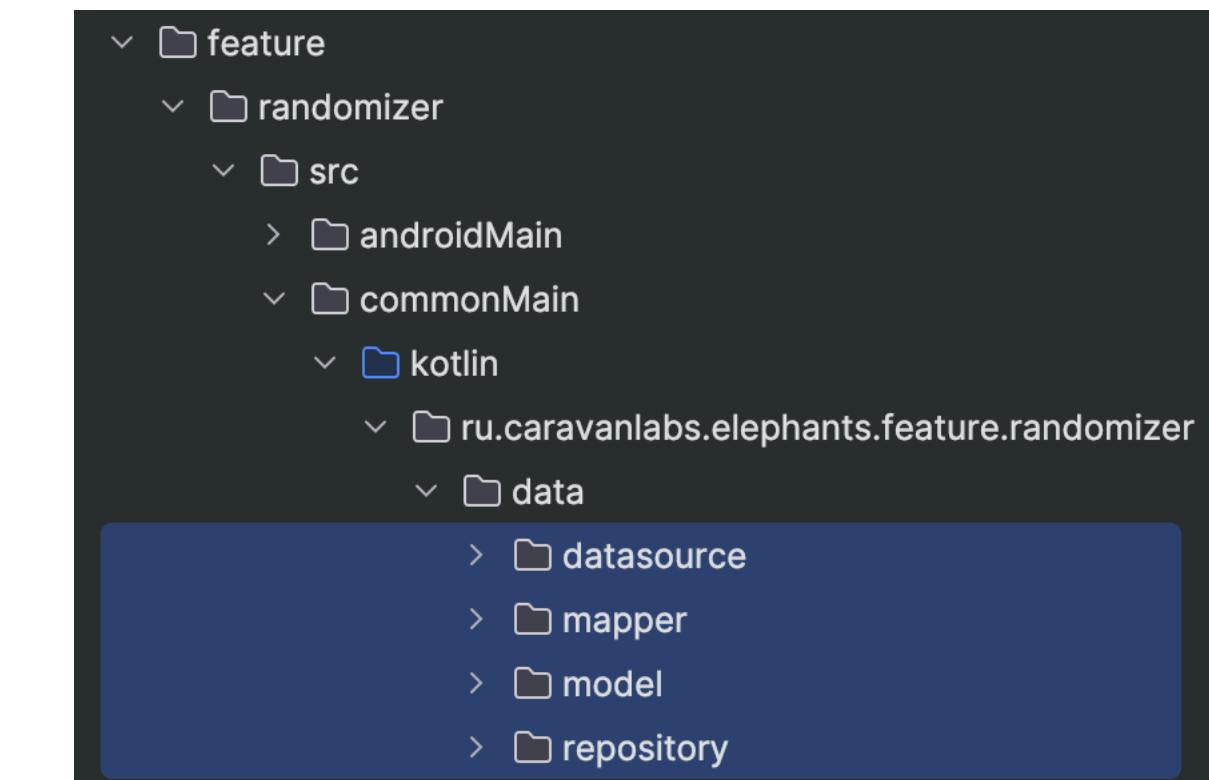
- Бизнес-модели как data class
- Интерфейс Repository
- UseCases для ключевых сценариев (например, GetRandomElephantUseCase)

7. Data-слой

Данный слой реализует доступ к данным, скрывая детали источников от Domain-слоя через интерфейсы репозиториев.

Компоненты:

- DTO модели - отражают структуру ответа API (аннотации `@Serializable`)
- DataSource - отвечает за работу с удалённым (Remote) или локальным источником
- Mapper - преобразует DTO в Domain Entity и обратно
- Repository Impl - содержит реализацию Domain-интерфейса, объединяя источники данных



Задание:

1. Реализуйте Data-слой с чётким разделением ответственности (`DTO → DataSource → Mapper → Repository`)
2. Подключите Ktor Client для сетевых запросов в `commonMain` (подробнее [здесь](#))
3. Для Android не забудьте добавить разрешение на интернет в манифест

8. Создание HttpClient

Задание: создать единый, сконфигурированный HTTP-клиент (Ktor) для всех платформ, инкапсулируя настройки (логирование, сериализация).

Шаги реализации:

1. В core/data/commonMain/network:

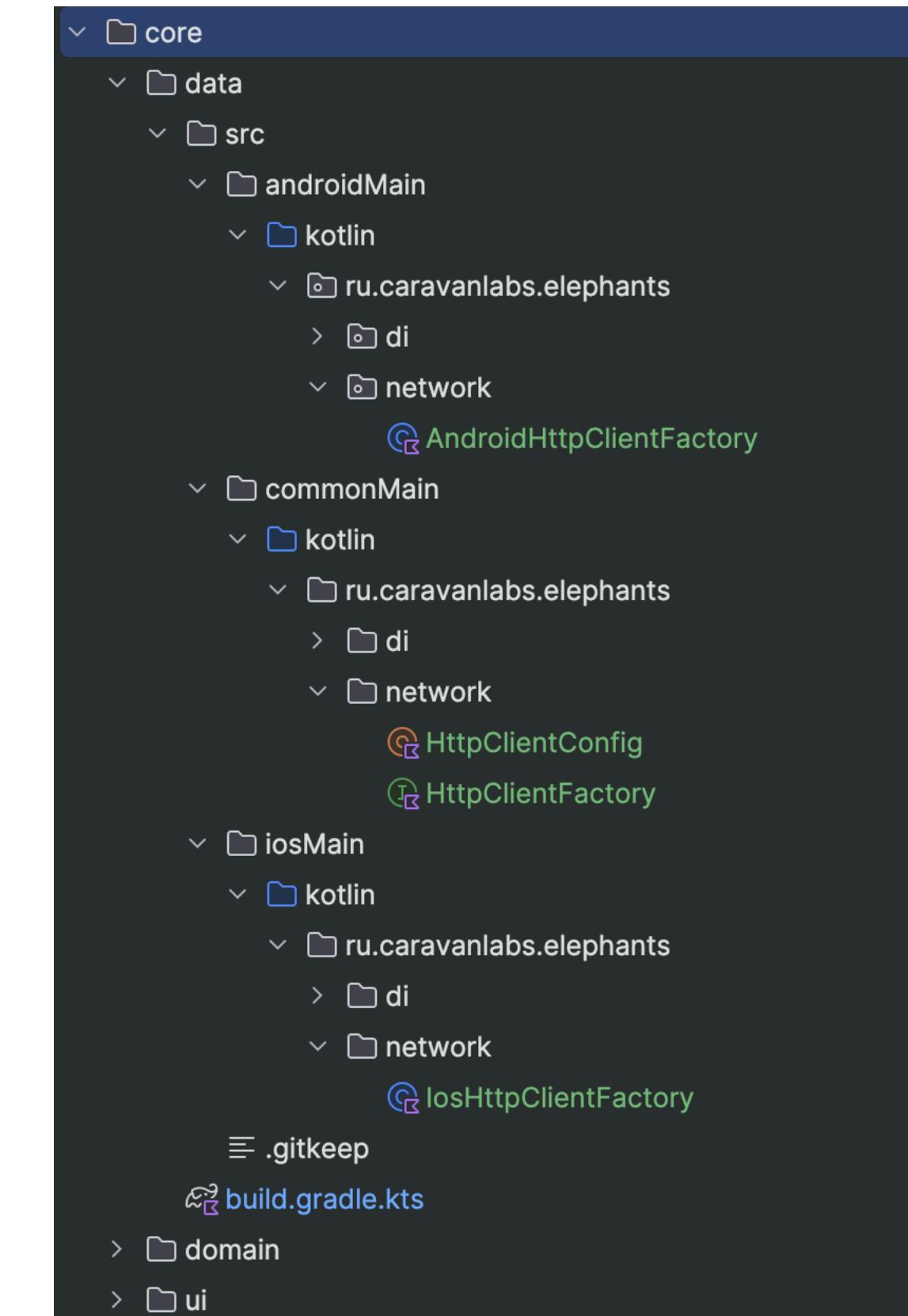
- Создайте interface HttpClientFactory с методом fun create(): HttpClient
- Создайте объект HttpClientConfig с методом configureClient(), который принимает HttpClient и добавляет общие плагины (ContentNegotiation, Logging через Napier)

2. В платформенных sourceSet'ах (androidMain, iosMain):

- Реализуйте фабрики (AndroidHttpClientFactory, IosHttpClientFactory)
- В фабриках используйте платформенный движок (например, OkHttp или Darwin) и примените общую конфигурацию через HttpClientConfig.configureClient()

Ключевое замечание

Перед началом убедитесь, что необходимые Ktor-зависимости (ktor-client-core, ktor-client-logging, ktor-serialization-kotlinx-json) подключены в build.gradle.kts модуля :core:data



9. Завершение реализации ViewModel

Теперь, когда Domain и Data слои готовы, дополните ViewModel рабочей бизнес-логикой

1. Интеграция с Domain-слоем

- Добавьте UseCase - внедрите зависимость от UseCase (например, GetRandomElephantUseCase) через конструктор
- Создайте метод-интенд - реализуйте метод loadRandomElephant(), который внутри будет вызывать useCase.invoke()

2. Обработка состояния

Метод loadRandomElephant() должен корректно обновлять State:

- Загрузка (isLoading = true) - перед вызовом UseCase
- Успех (elephant = data) - в случае успешного результата
- Ошибка (error = ...) - в случае исключения, с переводом в понятный для UI вид

10.1 Dependency Injection

Для связи всех компонентов и обеспечения тестируемости необходимо настроить Dependency Injection (DI) с использованием библиотеки Koin

1. Подключение Koin

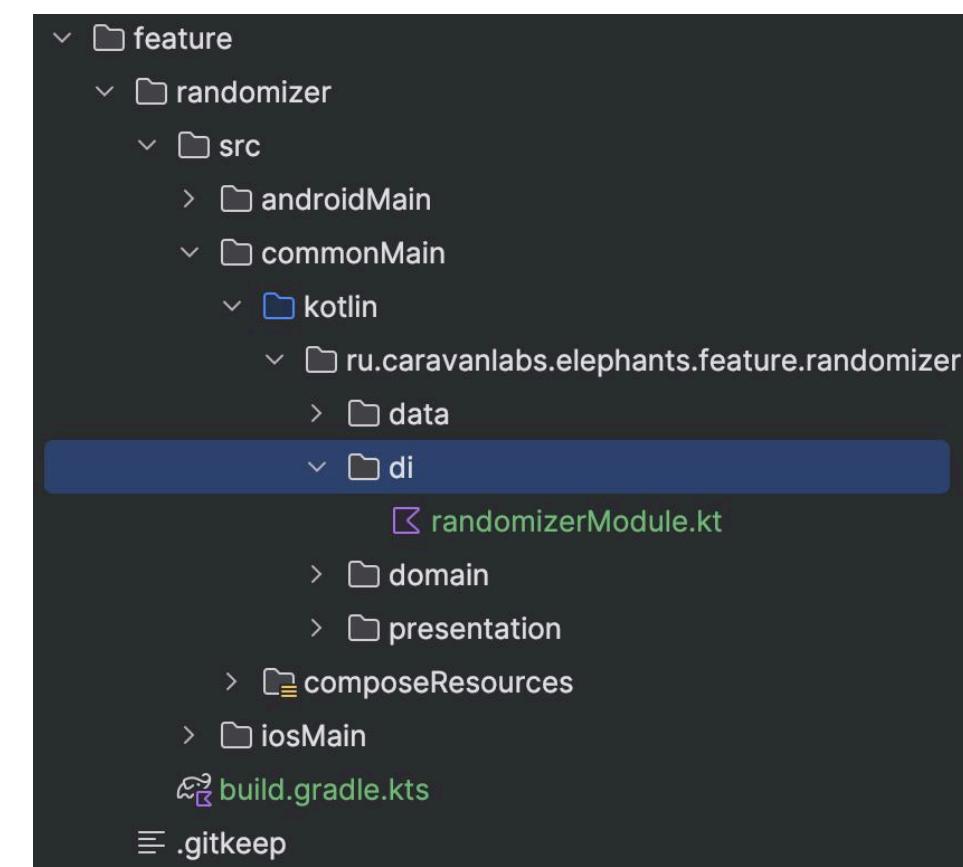
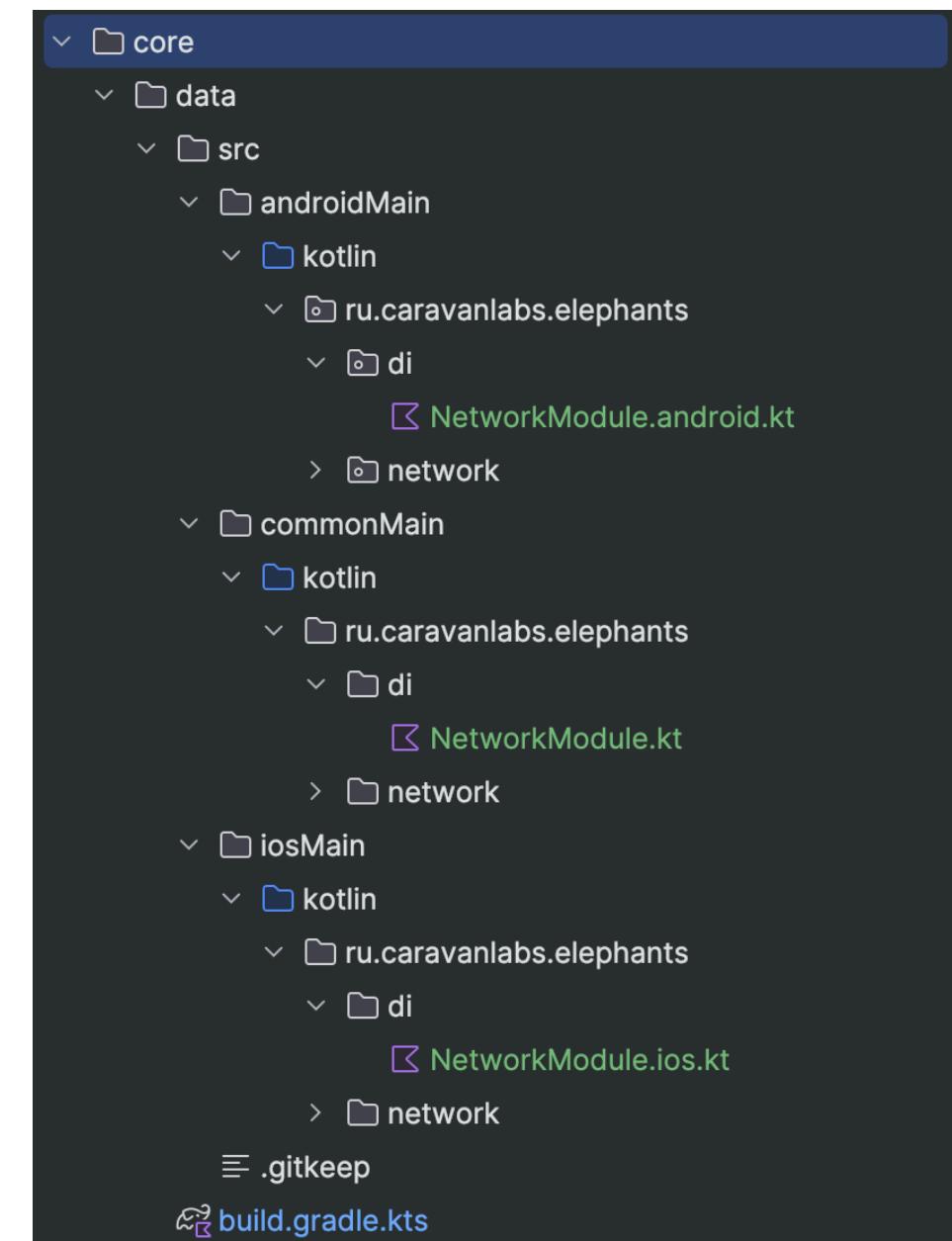
Добавьте зависимости Koin в build.gradle.kts модулей (:composeApp, :core, :feature) согласно [официальной документации](#)

2. Структура DI-модулей

1. Модуль core/data - создайте модуль для сетевых зависимостей, который объявляет HttpClient и фабрики для каждой платформы (см. Sample)
2. Модуль фичи (feature/.../di) - создайте в feature-модуле пакет di. В нём объявите все зависимости: RemoteDataSource, Mapper, Repository, UseCase и их реализации.
Не забудьте объявить ViewModel созданного экрана
3. Интеграция с экраном - в файле экрана используйте koinViewModel<RandomizerViewModel>() для получения instance ViewModel. Это заменит ручное создание viewModel() и обеспечит внедрение зависимостей

```
@Composable
fun RandomizerScreen() {

    val viewModel = koinViewModel<RandomizerViewModel>()
    val state by viewModel.state.collectAsState()
```



10.2 Инициализация DI

Чтобы все объявленные модули заработали, Koin необходимо инициализировать в точке входа приложения на каждой платформе

Места инициализации:

1. Android: В файле composeApp/androidMain/MainActivity, в методе onCreate
2. iOS: В файле composeApp/iosMain/MainViewController

```
startKoin {
    modules(
        // Перечислите здесь все модули
    )
}
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)
        startKoin {
            modules(...modules = randomizerModule, *getNetworkModules().toTypedArray())
        }
        setContent {
            App()
        }
    }

    @Preview
    @Composable
    fun AppAndroidPreview() {
        App()
    }
}
```

```
fun MainViewController() = ComposeUIViewController {
    startKoin {
        modules(...modules = randomizerModule, *getNetworkModules().toTypedArray())
    }
    App()
}
```

11. Compose Navigation

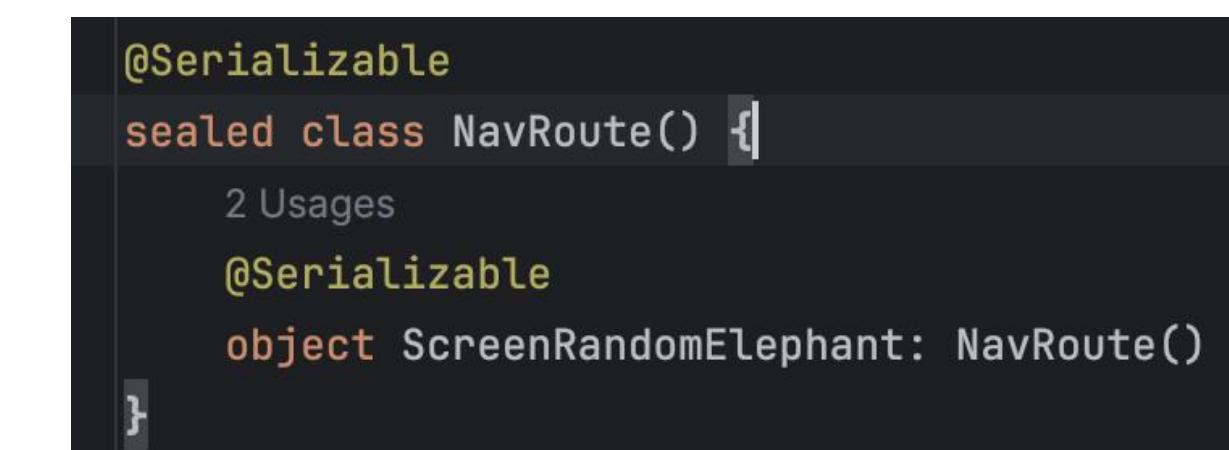
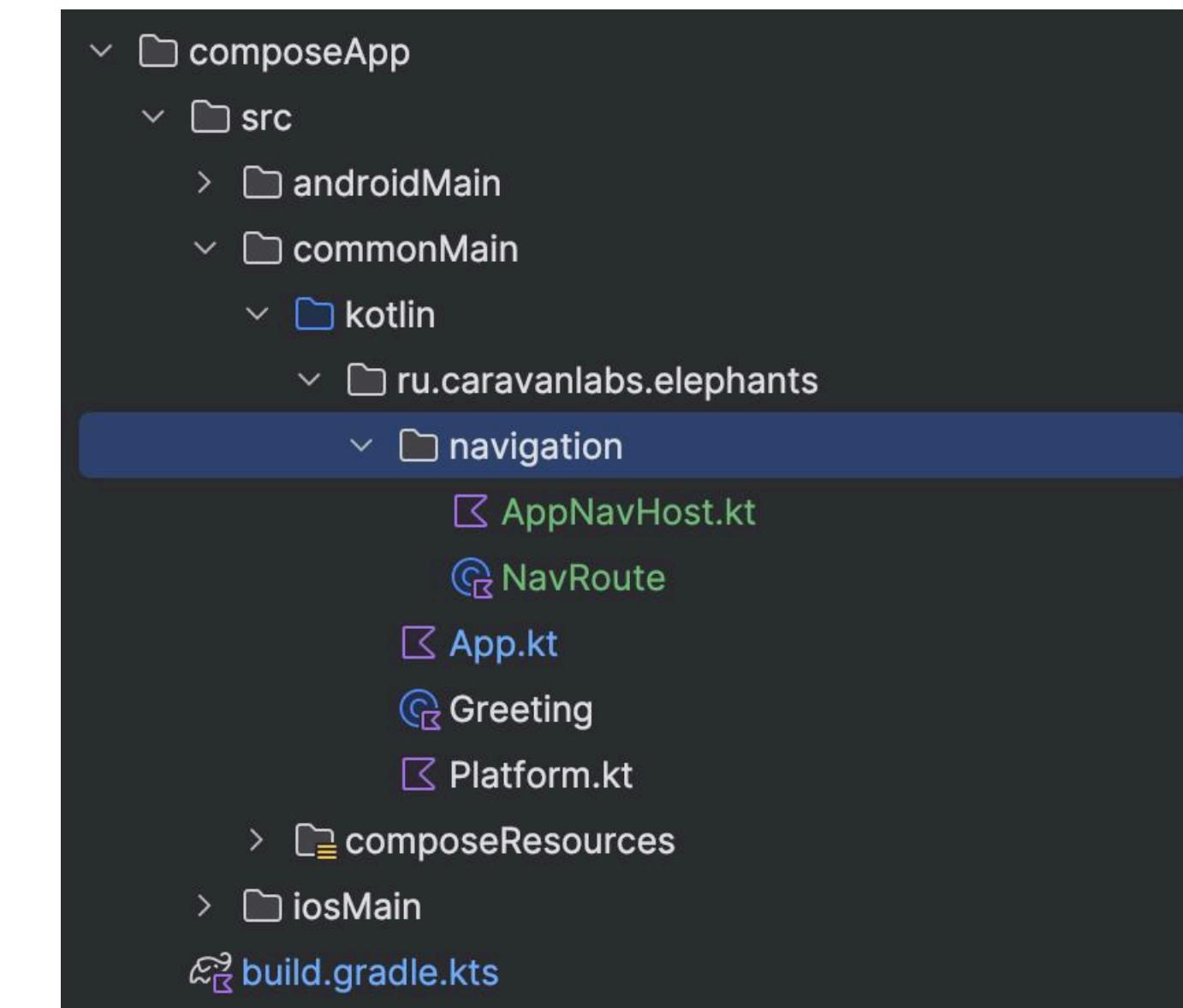
Для управления переходами между экранами в Compose Multiplatform используется библиотека Navigation

Шаги настройки:

1. Создайте пакет composeApp/commonMain/navigation.
2. Определите маршруты - создайте файл NavRoute.kt с sealed class (или object), реализующей NavGraphSpec из библиотеки.
Каждый экран является отдельным объектом
3. В файле AppNavHost.kt создайте @Composable функцию AppNavHost, которая принимает NavController
4. Внутри используйте NavHost, указывая startDestination и объявляя каждый экран в блоке composable { ... }
5. Создайте навигационный контроллер в главном @Composable приложения App:

```
val navController = rememberNavController()
```
6. В App передайте созданный контроллер в AppNavHost:

```
AppNavHost(navController = navController)
```



12. Реализация второй фичи

Задача: самостоятельно реализовать полноценную функциональность второго экрана «Тест: Какой ты слон?», применив на практике все изученные принципы KMP и Clean Architecture

Пояснения:

Повторите полный цикл разработки feature-модуля для нового экрана:

1. Создайте модуль quiz
2. Спроектируйте и реализуйте все три слоя архитектуры (Data, Domain, Presentation), включая:
 - Модели для вопросов, ответов и результата теста
 - Интерфейс и реализацию QuizRepository
 - UseCases для запуска теста и вычисления результата
 - QuizScreenState, QuizViewModel и UI на Compose
3. Настройте DI (Koin) для новой фичи
4. Интегрируйте в навигацию - добавьте новый маршрут в NavRoute и экран в AppNavHost

Успешное выполнение подтвердит ваше умение создавать модульные, поддерживаемые экраны в Kotlin Multiplatform!

Спасибо за участие!

Вы проделали отличную работу на практике. Теперь у вас есть готовый проект и понимание, как строить приложения на Kotlin Multiplatform. Продолжайте экспериментировать, задавать вопросы и создавать крутые мультиплатформенные продукты!

Удачи!

Буду рад вашему фидбеку и вопросам!

 sixvart05@bk.ru

 [@arterm_aweirro](https://t.me/arterm_aweirro)