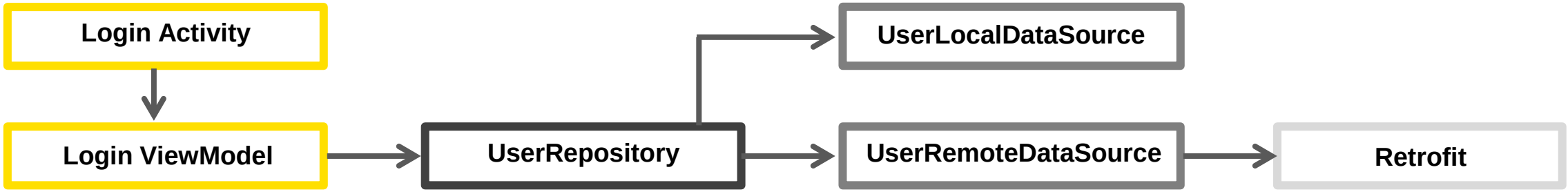
True Engineering

Проектирование современных приложений

Введение



Современные мобильные приложения требуют внимательного подхода к архитектуре и проектированию для обеспечения удобства разработки, масштабируемости и легкости поддержки.

В этой лекции рассмотрим ключевые аспекты проектирования современных Android-приложений, такие как модульность, **Dependency Injection (DI)** с использованием **Hilt** и **Koin**, а также использование **Jetpack Navigation Component** для организации навигации.

Основы проектирования современных приложений

- **Модульность**: Разделение приложения на независимые модули для упрощения разработки и тестирования.
- **Тестируемость**: Возможность легко тестировать различные компоненты приложения с минимальными зависимостями.
- **Поддерживаемость**: Легкость внесения изменений и добавления новых функций без значительных изменений в кодовой базе.
- **Производительность**: Оптимизация работы приложения для обеспечения высокой скорости и минимального энергопотребления.

Модульность

Модульность — это принцип проектирования приложений, который предполагает разделение приложения на независимые, переиспользуемые и легко поддерживаемые модули.

Зачем нужна модульность:

- **Упрощение разработки**: Модули могут разрабатываться и тестироваться независимо друг от друга.
- **Повышение повторного использования кода**: Модули могут использоваться в других проектах или повторно использоваться в рамках одного проекта.
- **Улучшение масштабируемости**: При добавлении новых функций достаточно изменить или добавить отдельные модули, не затрагивая остальной код

Основные принципы модульности

Single Responsibility Principle (SRP)

Каждый модуль должен отвечать за выполнение одной четко определенной задачи.

1

Инкапсуляция

Внутренние детали реализации модуля должны быть скрыты от других частей приложения. Взаимодействие между модулями должно происходить через четко определенные интерфейсы.

2

Слабая связанность

Модули должны минимально зависеть друг от друга. Это позволяет легко изменять или заменять один модуль, не влияя на работу других.

3

Высокая связность внутри

Все элементы внутри модуля должны быть логически связаны и работать над одной задачей.

4

Способы организации модульности

Feature Modules:

Разделение приложения на модули, каждый из которых отвечает за конкретную функциональность (например, аутентификация, профиль пользователя, покупки). Это упрощает добавление и поддержку новых функций.

Core Modules:

Создание отдельных модулей для общих библиотек и утилит, которые используются в разных частях приложения (например, работа с сетью, база данных).

Dynamic Feature Modules:

Модули, которые загружаются по требованию. Это позволяет уменьшить размер APK и загружать функциональность только тогда, когда она действительно нужна пользователю.

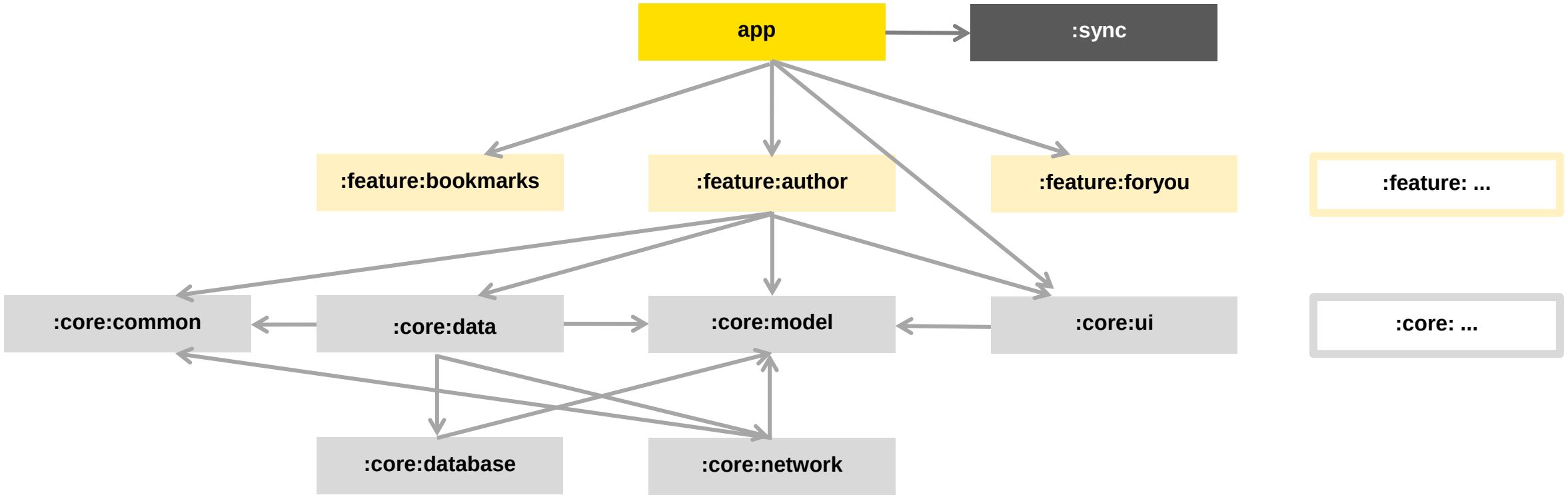
Пример: структура модульного приложения

Примерная структура модулей:

- **app**: Основной модуль, который связывает все остальные модули.
- **core**: Модуль с общими утилитами, которые используются в других модулях (например, логирование, работа с сетью).
- **feature-auth**: Модуль для функциональности аутентификации.
- **feature-profile**: Модуль для работы с профилем пользователя.
- **feature-payment**: Модуль для обработки платежей.
- **dynamic-feature-news**: Динамический модуль, загружаемый по требованию, для отображения новостей.

... и другие.

Пример: схематичное разделение на модули



Преимущества модульного подхода

- **Повышенная читаемость и поддерживаемость**: Код организован в независимые модули, что облегчает его понимание и поддержку.
- **Легкость тестирования**: Каждый модуль можно тестировать отдельно, что упрощает процесс автоматизированного тестирования.
- **Снижение времени сборки**: При модульной структуре можно собирать и компилировать только измененные модули, что ускоряет время сборки проекта.
- **Упрощение командной работы**: Модульная структура позволяет распределять задачи между разработчиками, минимизируя конфликты в коде.

Инструменты и подходы для реализации модульности

- **Gradle**: Использование мультипроектов в Gradle для управления зависимостями между модулями и упрощения сборки.
- **Dependency Injection**: Использование DI (например, Hilt или Koin) для управления зависимостями между модулями.
- **Jetpack Navigation Component**: Поддержка навигации между модулями и динамической загрузки модулей.

Dependency Injection (DI)

Dependency Injection (DI) — это техника, которая позволяет передавать зависимости объекта извне, что улучшает модульность и тестируемость кода.

Преимущества DI:

- Упрощение управления зависимостями.
- Улучшение тестируемости за счет возможности подмены зависимостей.
- Снижение связанности компонентов приложения.

Основные подходы к DI

Ручная реализация DI: Зависимости передаются через конструкторы или сеттеры. Подходит для небольших приложений, но не масштабируется.

Использование Dagger/Hilt: Популярный фреймворк для DI, предлагающий высокую производительность и автоматическое управление зависимостями.

Использование Koin: Легковесный и простой в использовании DI-фреймворк, не требующий аннотаций и генерации кода.

Введение в Hilt

Hilt — это библиотека для **Dependency Injection**, разработанная на базе **Dagger** и адаптированная для Android.

Основные преимущества **Hilt**:

- Упрощает настройку Dagger в Android-приложениях.
- Обеспечивает удобную интеграцию с жизненным циклом Android-компонентов (Activity, Fragment и т.д.).
- Предоставляет автоматическую поддержку AndroidViewModel, WorkManager и других компонентов.

Пример: использование Hilt в приложении

build.gradle — Kotlin

```kotlin
dependencies {
    implementation "com.google.dagger:hilt-android:2.44"
    kapt "com.google.dagger:hilt-android-compiler:2.44"
}
```

MainActivity.kt — Kotlin

```kotlin
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    @Inject lateinit var myDependency: MyDependency

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // Использование myDependency
    }
}
```

MyApplication.kt — Kotlin

```kotlin
@HiltAndroidApp
class MyApplication : Application()
```

AppModule.kt — Kotlin

```kotlin
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    @Singleton
    fun provideMyDependency(): MyDependency {
        return MyDependencyImpl()
    }
}
```

Введение в Koin

Koin — это легковесный **DI-фреймворк**, который не требует генерации кода и легко интегрируется в Android-приложения.

Преимущества **Koin**:

- Простой синтаксис и легкость настройки.
- Возможность создания модулей в виде простых функций.
- Поддержка внедрения зависимостей во ViewModel и других компонентах.

Пример: использование Koin в приложении

build.gradle — Kotlin

```kotlin
dependencies {
    implementation "io.insert-koin:koin-android:3.4.0"
}
```

MainActivity.kt — Kotlin

```kotlin
class MainActivity : AppCompatActivity() {
    private val myDependency: MyDependency by inject()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // Использование myDependency
    }
}
```

MyApplication.kt — Kotlin

```kotlin
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@MyApplication)
            modules(appModule)
        }
    }
}
```

AppModule.kt — Kotlin

```kotlin
val appModule = module {
    single<MyDependency> { MyDependencyImpl() }
}
```

Сравнение Hilt и Koin

Hilt:

- Подходит для крупных проектов, требующих высокой производительности.
- Автоматическое управление жизненным циклом зависимостей.
- Сложность настройки из-за необходимости генерации кода.

Koin:

- Легковесный и простой в использовании, идеально подходит для небольших и средних проектов.
- Не требует аннотаций и генерации кода.
- Простой синтаксис, но возможна потеря производительности на крупных проектах.

Введение в Jetpack Navigation Component

Jetpack Navigation Component — это библиотека, упрощающая управление навигацией внутри приложения, обеспечивая единообразие и гибкость.

Основные возможности:

- Декларативное описание навигационных маршрутов.
- Управление стеком навигации и поддержка глубоких ссылок (Deep Links).
- Интеграция с компонентами жизненного цикла (Lifecycle) и ViewModel.
- Поддержка анимации и переходов

Основные компоненты Navigation Component

- **NavController**: Управляет навигацией между экранами.
- **NavGraph**: Описывает маршруты и связи между экранами.
- **NavHostFragment**: Контейнер, который управляет навигацией внутри фрагментов.
- **NavArgs**: Позволяет передавать данные между фрагментами через безопасные типизированные аргументы.

Пример: использование Navigation Component (1)

build.gradle — Kotlin

```
dependencies {
    implementation "androidx.navigation:navigation-fragment-ktx:2.7.1"
    implementation "androidx.navigation:navigation-ui-ktx:2.7.1"
}
```

activity_main.xml — XML

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true" />
```

nav_graph.xml — XML

```
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    app:startDestination="@id/homeFragment">

    <fragment
        android:id="@id/homeFragment"
        android:name="com.example.app.HomeFragment"
        tools:layout="@layout/fragment_home" >
        <action
            android:id="@id/action_homeFragment_to_detailFragment"
            app:destination="@id/detailFragment" />
    </fragment>

    <fragment
        android:id="@id/detailFragment"
        android:name="com.example.app.DetailFragment"
        tools:layout="@layout/fragment_detail" />
</navigation>
```

Пример: использование Navigation Component (2)

MainActivity.kt — Kotlin

```kotlin
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val navController = findNavController(R.id.nav_host_fragment)
        val appBarConfiguration = AppBarConfiguration(navController.graph)
        setupActionBarWithNavController(navController, appBarConfiguration)
    }

    override fun onSupportNavigateUp(): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return navController.navigateUp() || super.onSupportNavigateUp()
    }
}
```

Продвинутые возможности Navigation Component

- **Deep Links**: Позволяют запускать приложение на определенном экране из внешних источников, таких как браузер или другое приложение.
- **Navigation Safe Args**: Генерирует безопасные типизированные аргументы для передачи данных между фрагментами.
- **Dynamic Feature Modules**: Поддержка навигации между модулями, загружаемыми по требованию, что позволяет уменьшить размер APK и ускорить установку приложения.
- **Custom Transitions**: Возможность создания и использования пользовательских анимаций и переходов между экранами.

Пример: реализация DeepLink с использованием Navigation Component

nav_graph.xml — XML

```xml
<fragment
    android:id="@id/detailFragment"
    android:name="com.example.app.DetailFragment"
    tools:layout="@layout/fragment_detail" >
    <deepLink
        android:id="@+id/deepLink"
        app:uri="http://www.example.com/details/{itemId}" />
</fragment>
```

DeepLinkCall.kt — Kotlin

```kotlin
val deepLinkUri = Uri.parse("http://www.example.com/details/123")
val deepLinkIntent = Intent(Intent.ACTION_VIEW, deepLinkUri)
startActivity(deepLinkIntent)
```

MainActivity.kt — Kotlin

```kotlin
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val navController = findNavController(R.id.nav_host_fragment)
        if (intent?.data != null) {
            navController.handleDeepLink(intent)
        }
    }
}
```

Jetpack Navigation Component VS традиционные методы

Jetpack Navigation Component

Преимущества:

- Декларативный подход к описанию маршрутов.
- Поддержка DeepLink и безопасной передачи данных.
- Интеграция с жизненным циклом компонентов.

Недостатки:

- Может потребоваться время на изучение и настройку.
- Менее гибок по сравнению с ручной навигацией в редких и сложных сценариях.

Традиционные методы

Преимущества:

- Полный контроль над процессом навигации.
- Гибкость в реализации кастомных сценариев.

Недостатки:

- Более сложное управление стеком навигации.
- Риск возникновения ошибок при передаче данных между экранами.

Заключение

- **Проектирование современных приложений** требует использования передовых инструментов и подходов для обеспечения удобства разработки, тестируемости и поддержки кода.
- **Модульность** способствует улучшению организации кода, его поддерживаемости и масштабируемости.
- **Dependency Injection** с использованием **Hilt** или **Koin** упрощает управление зависимостями, повышая модульность и тестируемость приложения.
- **Jetpack Navigation Component** предоставляет мощные средства для организации навигации внутри приложения, упрощая работу с глубокими ссылками, анимациями и управлением стеком навигации.

True Engineering

True Engineering

630128, г. Новосибирск,
ул. Кутателадзе, 4г

(383) 363-33-51, 363-33-50
info@trueengineering.ru
trueengineering.ru

Новосибирский Государственный Университет