

# Missing Parts?

А так можно было?

Клещин Никита

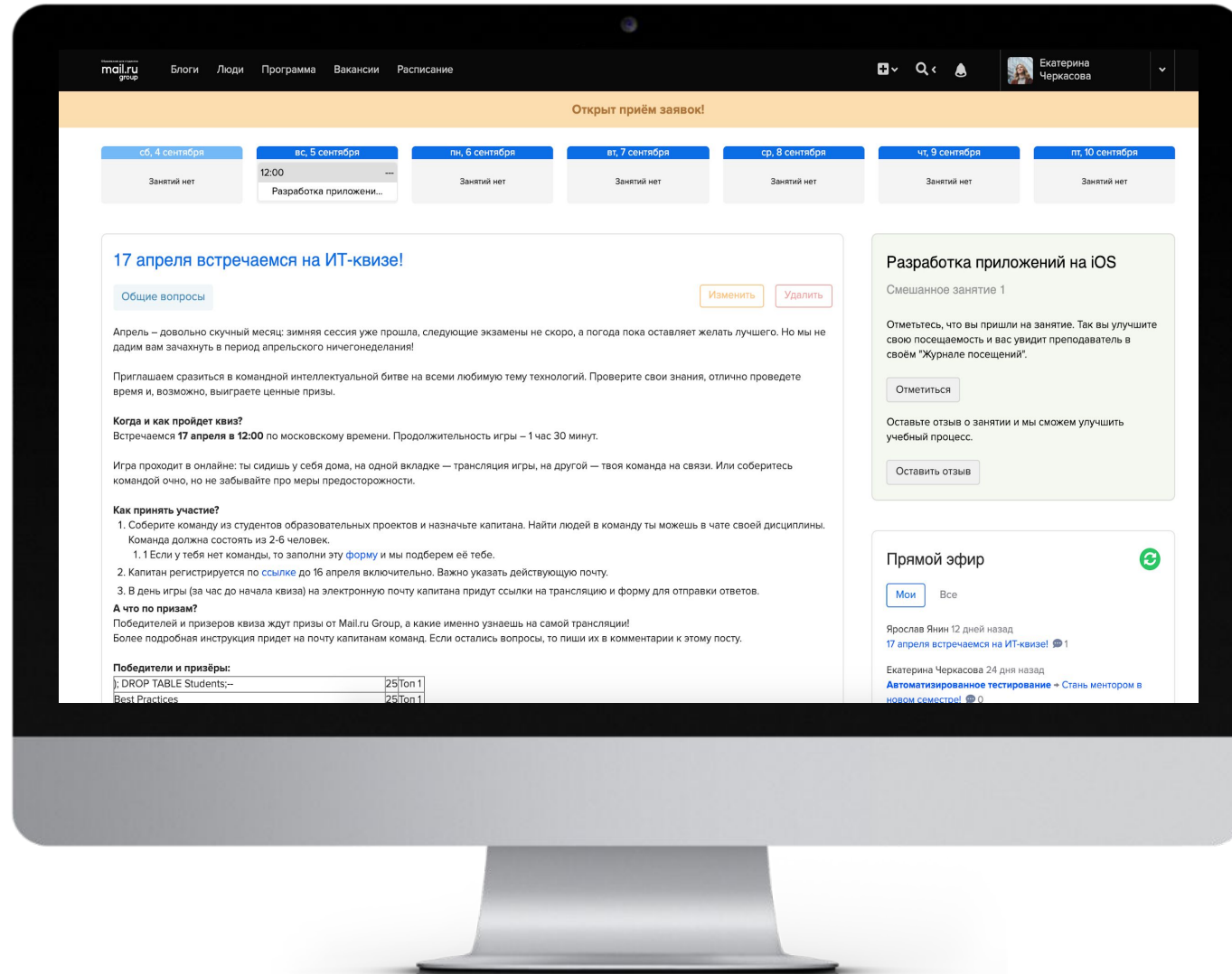




# Содержание занятия

1. О чем еще можно рассказать?
2. ???
3. ???
4. ???
5. ???

# Не забудьте отметиться!



О чем еще можно  
рассказать?





**А каких тем не  
хватает вам?**

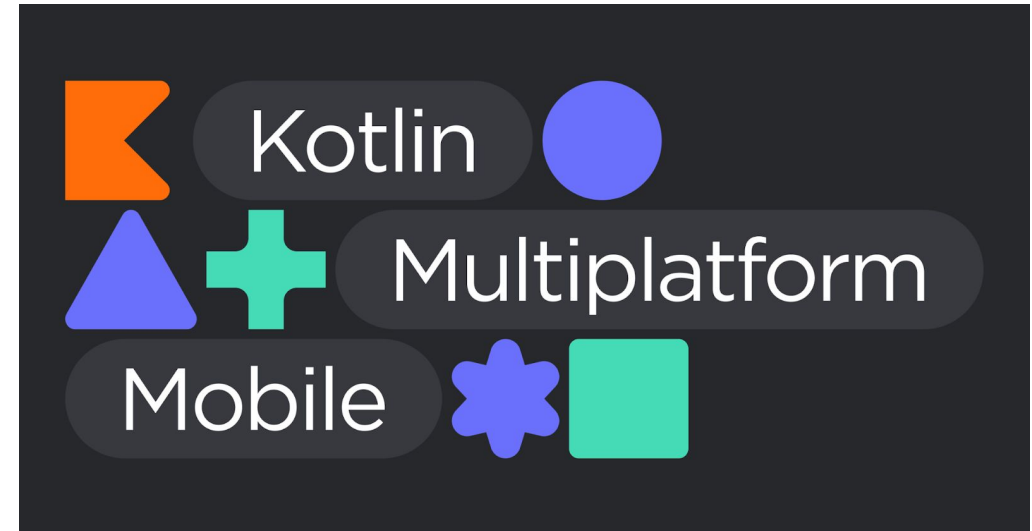
# Кроссплатформенность

Эта тема последнее время набирает большую популярность, и уже есть компании, которым важно, чтобы у людей были необходимые навыки.

Но такая тема имеет сложности:

- В рамках лекции нормально рассказать пока нет возможности
- Желательно иметь Mac и написать приложение под iOS (чтобы проверить возможность реализации)

Темы для ознакомления - **Flutter**, **KMP**, **KMM**



# AndroidTV и Wearable

Это достаточно специфичные “платформы”. Они основаны на Android API, но имеют свои нюансы.

Под AndroidTV писать на чистых View или Compose - достаточно сложно и долго (из-за особенности работы фокусов и переключений).

Основной фреймворк для создания приложений под AndroidTV - это **Leanback**

<https://developer.android.com/training/tv/start>

Популярность этих платформ среди пользователей - очень низкая. Но некоторые приложения скорее актуальны именно для них.

# androidtv



# Релизы в магазины

**GooglePlay** для нас остается важным магазином для распространения. Но есть пользователи, которые пользуются и тем, что предоставляет им вендор самого устройства.

Магазины, на которые еще стоит обратить внимание - **AppGallery** (от Huawei), **MiStore**. И следить за новостями о новых российских магазинах.

Есть еще специфические магазины под телевизоры.





# Android и WebView





Он может  
пригодиться?

# Какие для него есть варианты

- Отобразить какую-то дополнительную (второстепенную) информацию;
- Отображать информацию, которая должна быть изменяемой;
- Имеющийся опросник;
- Форма оплаты; (?)
- Делаем приложение на основе адаптивного сайта; (?)
- Общаться с JS (?)



# Очень прост в настройке

**WebView** - View для рендеринга страницы. Браузер внутри приложения.

В коде можно еще донстроить WebView под свои нужды (включить JavaScript или сделать ручную обработку переходов)

Включает в себя дефолтные контролы и обработчики для зума, а навигация (forward, back) должна будет делаться исключительно своими силами:)

```
// manifest
<uses-permission android:name="android.permission.INTERNET" />

// layout
<WebView
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>

// load page
webview.loadUrl(url)

// advanced
webview.settings.apply {
    javaScriptEnabled = true
}

webview.apply {
    webViewClient = CustomWebClient()
}
```

# Собственная обработка переходов

```
webView.apply {  
    webViewClient = SandboxWebClient()  
}  
  
class SandboxWebClient: WebViewClient() {  
    override fun shouldOverrideUrlLoading(view: WebView?, url: String?): Boolean {  
        if (isForbiddenUrl(url)) {  
            Toast.makeText(view!!.context, "Forbidden Url: $url", Toast.LENGTH_LONG).show()  
            return true  
        }  
  
        return false  
    }  
  
    protected fun isForbiddenUrl(url: String?): Boolean {  
        return !url.isNullOrEmpty() && url.startsWith("http")  
    }  
}
```

Немного практики

# Взаимодействие Android с функциями страницы

Проще всего построить взаимодействие при помощи вызова JavaScript функций самой страницы. Код самой функции уже может делать изменения на странице.

Необходимо разрешить использование JavaScript.

Для вызова JavaScript-функции поможет метод **evaluateJavascript**. Где первый параметр - это вызов самой функции. Второй параметр - подписка на результат исполнения

Если поискать еще варианты вызова, для старых Android работал такой вариант **webView.loadUrl("javascript:enable();")**.

```
// !!!!
webView.settings.apply {
    javascriptEnabled = true
}
```

```
// without result
webView.evaluateJavascript("function1()", null)
```

```
// with result
webView.evaluateJavascript("function2('$param')") {
    // return String
}
```

# Взаимодействие страницы с Android

Необходимо разрешить использование JavaScript.

Зарегистрировать свой **JavascriptInterface** при помощи метода **addJavascriptInterface**. Вторым параметром метода, **name**, потребуется для того, чтобы дергать методы именно этого интерфейса со страницы.

**JavascriptInterface** класс создается без наследования. Но методам, к которым будет обращаться JavaScript, надо будет пометить аннотацией **@JavascriptInterface**.

```
// !!!!
webview.settings.apply {
    javaScriptEnabled = true
}

// set callback
webview.apply {
    addJavascriptInterface(CustomWebInterface(), "Android")
}

// callback example
class CustomWebInterface(val activity: Activity) {
    @JavascriptInterface
    fun showToast(message: String) {
        ...
    }
}

// on page
function showAndroidToast(message) {
    Android.showToast(message);
}
```

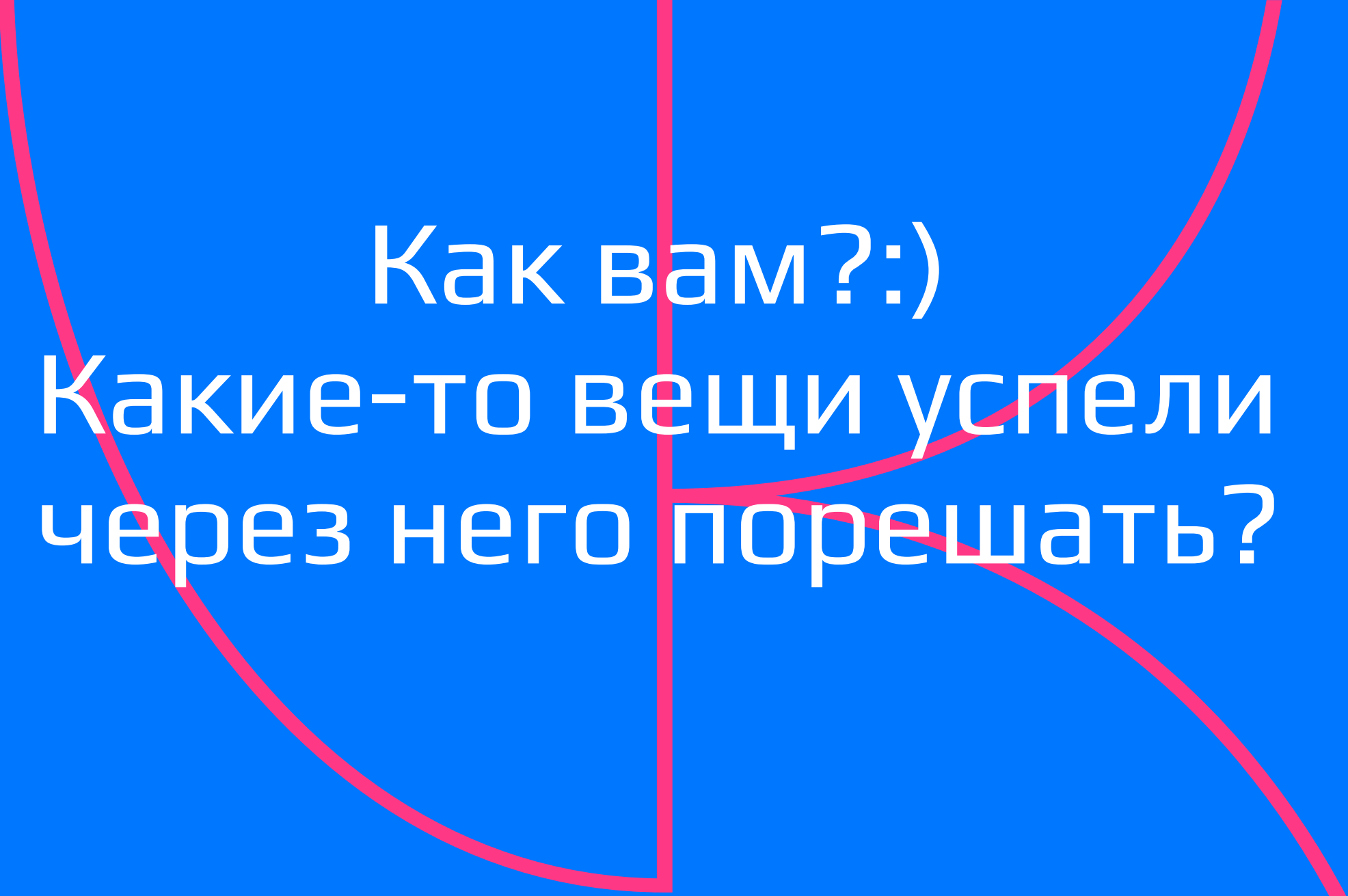


Как это в проекте

# Немного о Gradle

Точнее, об Android Gradle Plugin





Как вам?:)  
Какие-то вещи успели  
через него порешать?

# Что в нем еще есть полезного

**Android Gradle Plugin** дает достаточно много инструмента для решения разных проблем.

Но зачастую про них начинаем узнавать, когда сталкиваемся с проблемами, которые уже на уровне кода решить нет возможности

```
// apk signing
signingConfigs { }

// build type variants
buildTypes { }

// yet another build variants
flavorDimensions "structure", "vendor"
productFlavors { }

// dependency conflict resolving...
packagingOptions { }

// additional repositories (not AGP)
repositories { }

// just variables (not AGP)
buildscript {
    ext.kotlin_version = '1.5.31'

// additional variables for BuildConfig
buildConfigField "boolean", "DEBUG_CONFIG", "true"

// additional resource value
resValue "string", "base_url", "https://sample.ru"
```

# Подпись?

Она важна именно для релизных сборок.

Подпись делает так, что поверх вашего приложения не встанет приложение с таким же именем пакета, если у него отличная подпись.

Сейчас в GooglePlay запрещено распространение APK, все должны переходить на формат AAB, который при превращении в APK подпишет сам магазин.

Но еще остались магазины, которые позволяют выгружать APK.

```
signingConfigs {  
    dummy {  
        storeFile file("$rootDir/danger/dummy.jks")  
        storePassword 'qwerty'  
        keyAlias 'dummy'  
        keyPassword 'qwerty'  
    }  
}  
  
buildTypes {  
    debug {  
        signingConfig signingConfigs.dummy  
    }  
  
    release {  
        signingConfig signingConfigs.dummy  
    }  
}
```

# Варианты билдов

По дефолту есть два типа - **release** и **debug**.

Но по потребностям можно будет добавить свой тип (который для удобства может основываться на основном типе) и в нем провести дополнительные настройки.

Здесь основные флаги:

- **signingConfig** - чем подписывать приложение
- **debuggable** - возможность дебажить этот тип
- **minifyEnabled** - оптимизация веса приложения
- **proguardFiles** - настройка обфускации

```
buildTypes {  
    debug {  
        signingConfig signingConfigs.dummy  
    }  
  
    release {  
        signingConfig signingConfigs.dummy  
        minifyEnabled false  
    }  
  
    unsafe {  
        initWith release  
        debuggable true  
        matchingFallbacks = ['release']  
    }  
}
```

# “Вкусы” приложения

Из примера справа видно, что есть два вида “вкусов”. При желании их можно делать и больше.

Оно полезно, когда мы билдим приложения в разные магазины. И мы хотим на уровне компиляции разные библиотеки положить в приложение, чтобы не тащить весь код.

Как пример, ранее показывали, что при помощи “вкусов” можно сделать trial и payment версии приложений в рамках одного проекта.

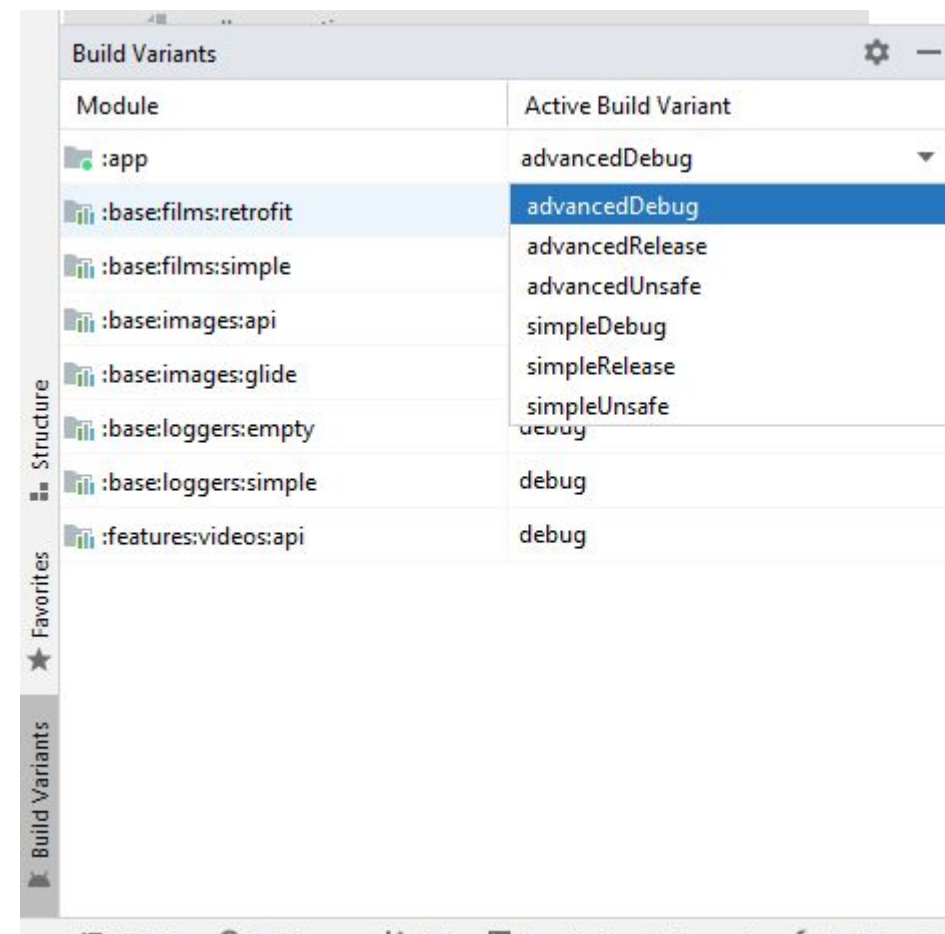
```
flavorDimensions "structure", "vendor"
productFlavors {
    simple {
        dimension "structure"
    }

    advanced {
        dimension "structure"
    }

    huawei {
        dimension "vendor"
    }

    google {
        dimension "vendor"
    }
}
```

# Результат таких настроек





# “Вкусы” и типы позволяют вам конфигурировать зависимости

```
dependencies {  
    implementation project(":base:loggers:api")  
    debugImplementation project(":base:loggers:simple")  
  
    simpleImplementation 'ru.tinkoff.decoro:decoro:1.5.0'  
  
    advancedReleaseImplementation project(":base:loggers:empty")  
  
    testImplementation 'junit:junit:4.+'  
    testImplementation "io.mockk:mockk:1.12.3"  
  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
  
    ...  
}
```

# Дополнительные значения

- **buildConfigField** положит значение в **BuildConfig** файл модуля
- **resValue** положит ресурс в values модуля

**resValue** придется использовать, если нет возможности подложить эти значения стандартными средствами gradle - это создание папки с нужным типом или “вкусом”, на уровне папки **main**, и там по Android структуре проекта создание res папки и папки **values**.

```
flavorDimensions "structure"
productFlavors {
    simple {
        dimension "structure"

        buildConfigField "boolean", "IS_SIMPLE", "true"
        resValue "string", "base_url", "https://simple.ru"
    }
}

buildTypes {
    release {
        buildConfigField "boolean", "DEBUG_CONFIG", "false"
        resValue "string", "analytics_url", "https://..."
    }
}
```

# Конфликты из-за одинаковых файлов

Это не частый кейс.

Но иногда при затаскивании зависимостей, gradle может ругаться, что есть какие-то одинаковые файлы.

В данном случае можно при помощи этой опции указать, как это надо будет обработать.

Чаше встречается проблема, когда тащатся одинаковые файлы и кладутся по одному пути. Если файлы не важные для работы кода - их просто удаляют.

```
packagingOptions {  
    exclude 'META-INF/INDEX.LIST'  
    exclude ("META-INF/*.kotlin_module")  
}
```

# Репозитории

Этот пункт позволяет управлять репозиториями, которые используются для выкачивания зависимостей.

Не все библиотеки выкладывают в публичные репозитории. Какие-то хранятся на внутренних репозиториях под авторизацией и без.

```
repositories {  
    maven {  
        credentials {  
            username System.getenv('PREMIER_NEXUS_LOGIN')  
            password System.getenv('PREMIER_NEXUS_PASSWORD')  
        }  
  
        url System.getenv('PREMIER_NEXUS_MAVEN')  
    }  
  
    maven { url = 'https://developer.huawei.com/repo/' }  
    maven { url = 'https://dl.google.com/dl/android/maven2/' }  
    mavenCentral()  
}
```

# Глобальные переменные для сборки

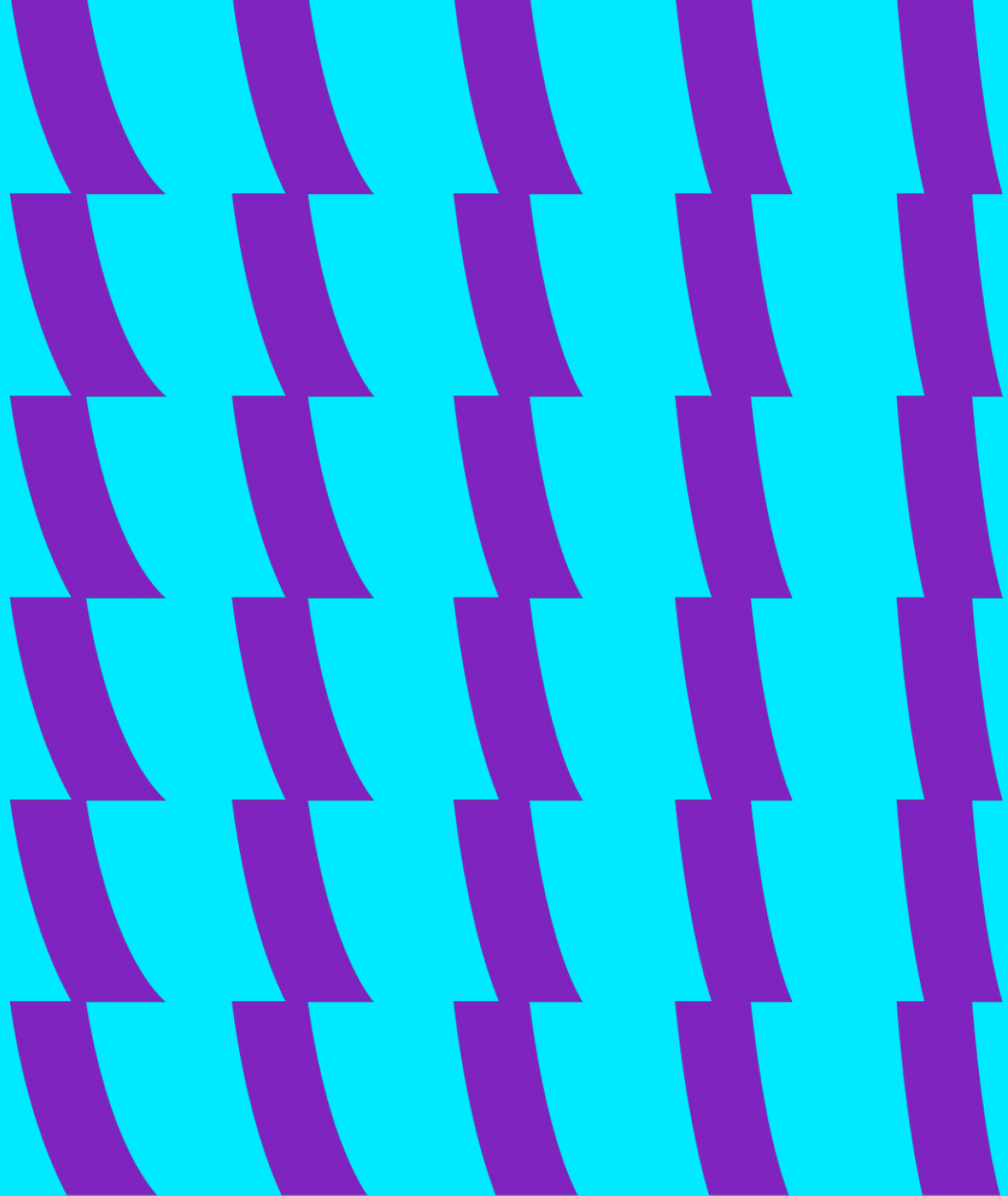
Эти переменные можно заносить разными способами.

Обычно используется для версии библиотек. В данном случае, если версия будет устаревшей, то gradle ее даже подсветит в конфигах

```
buildscript {  
    ext.kotlin_version = '1.5.31'  
    ext.coroutines_version = '1.5.2'  
    ...  
}  
  
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"  
    ...  
}
```

# Модули?

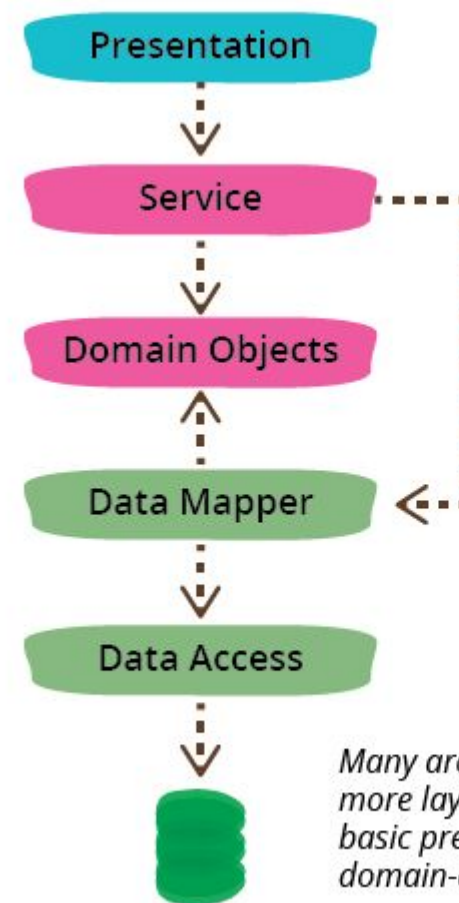
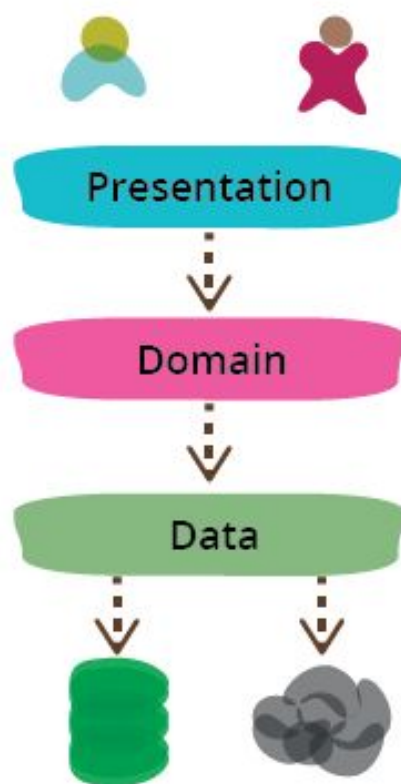
Монолит vs Многомодульность





Что это такое?  
Для чего это нужно?

# Одна из наиболее распространенных схем приложения



*Many architectures with more layers follow the basic presentation-domain-data pattern*

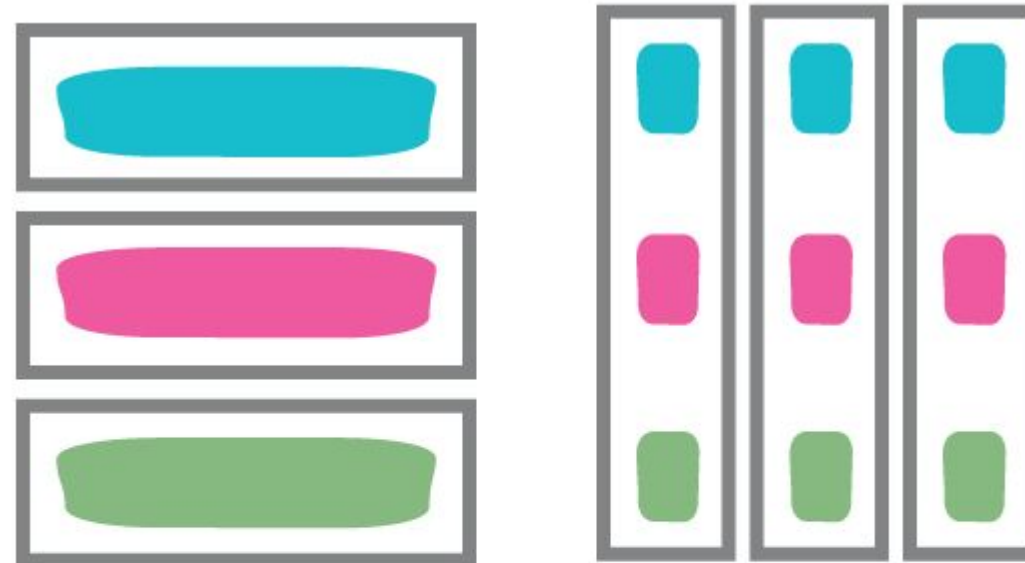


# Один из вариантов построения модулей

Тут я бы поддержал идею Мартина Фаулера - модули стоит делать “фуллстэковыми”

<https://martinfowler.com/bliki/PresentationDomainDataLayering.html>

Основной довод - это позволит модулю расти абстрагированно. И в случае чего - быть заменяемым, без чистки других модулей.



*Don't use layers as the top level modules in a complex application...*



*... instead make your top level modules be full-stack*

# Как выглядит модульность в Android проекте?

При создании проекта у нас уже появляется основной модуль - **app**.

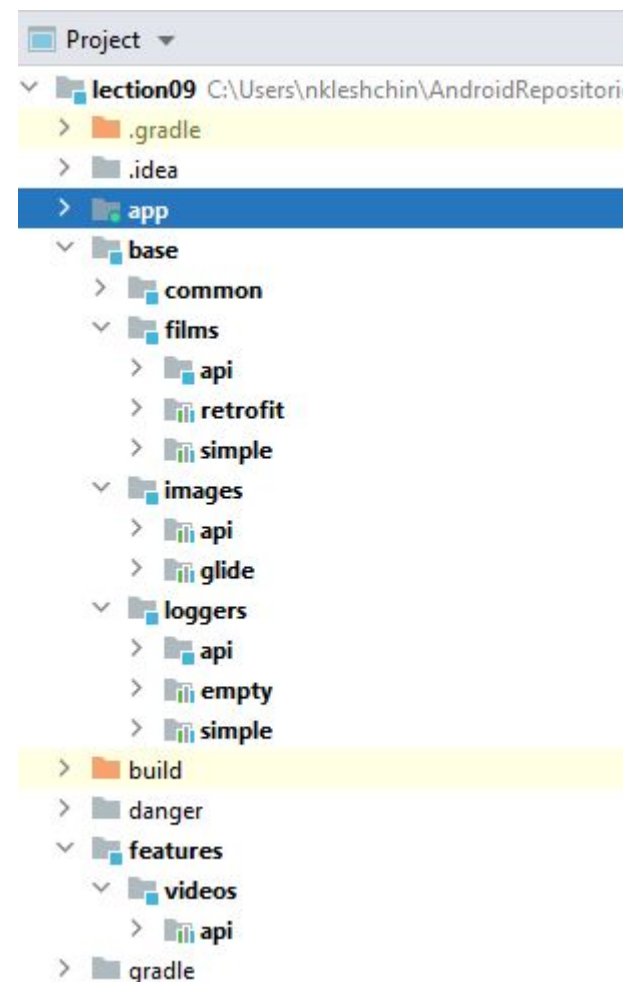
Другие модули строятся по той же логике. Отличие только в плагине, который надо будет применить в build.gradle файле, на уровне модуля.

Вместо плагина для приложения:

- `apply plugin: 'com.android.application'`

Надо будет применить один из следующих плагинов:

- `apply plugin: 'com.android.library'`
- `apply plugin: 'java-library'`



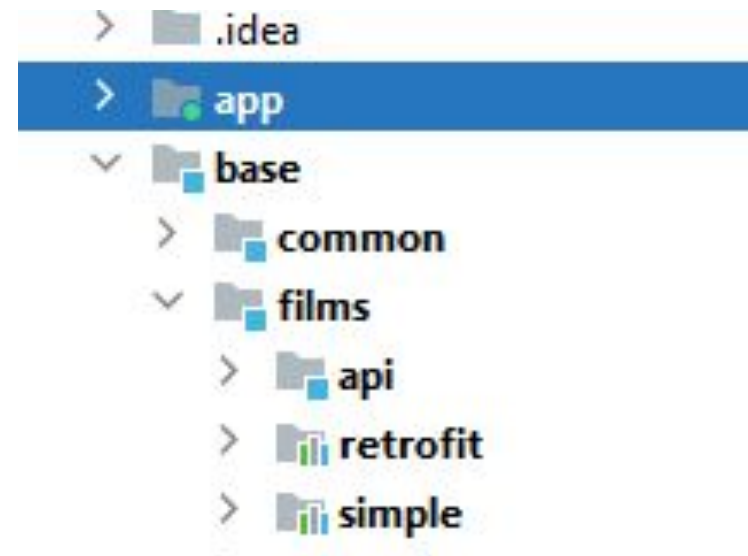
# Как построить модуль?

Зависит, конечно, от потребностей.

Один из хороших подходов - это делить модуль на **API** и **IMPLEMENTATION**. Но это подходит для сложных реализаций

Где

- **API** - будет содержать все необходимые объекты для создания реализации. Этот модуль можно будет подключать к другим модулям, которым важно именно работать с ним, но не важна имплементация (?).
- **IMPLEMENTATION** - вариант реализации. Он наследует **API** модуль и для его компонентов делает необходимые реализации. В результате этот модуль можно будет добавлять только в конечное приложение.



# Как добавлять локальные модули в проект

Используется все тот же механизм, что и с зависимостями. Только мы вместо ссылки на зависимость делаем ссылку на “проект”.

```
dependencies {  
    ...  
    implementation project(":base:common")  
  
    implementation project(":base:loggers:api")  
    debugImplementation project(":base:loggers:simple")  
    unsafeImplementation project(":base:loggers:simple")  
    releaseImplementation project(":base:loggers:empty")  
  
    implementation project(":base:images:api")  
    implementation project(":base:images:glide")  
  
    implementation project(":features:videos:api")  
    simpleImplementation project(":base:films:simple")  
    advancedImplementation project(":base:films:retrofit")  
}
```

# Другие ключевые слова для добавления зависимостей

**api** - работает как `implementation`, но позволяет верхнеуровненному модулю получить доступ к объектам добавляемого модуля

**compileOnly** - добавляет модуль только для компиляции. Используется обычно для того, чтобы просто собрать приложение, и не тянуть в него дополнительные зависимости

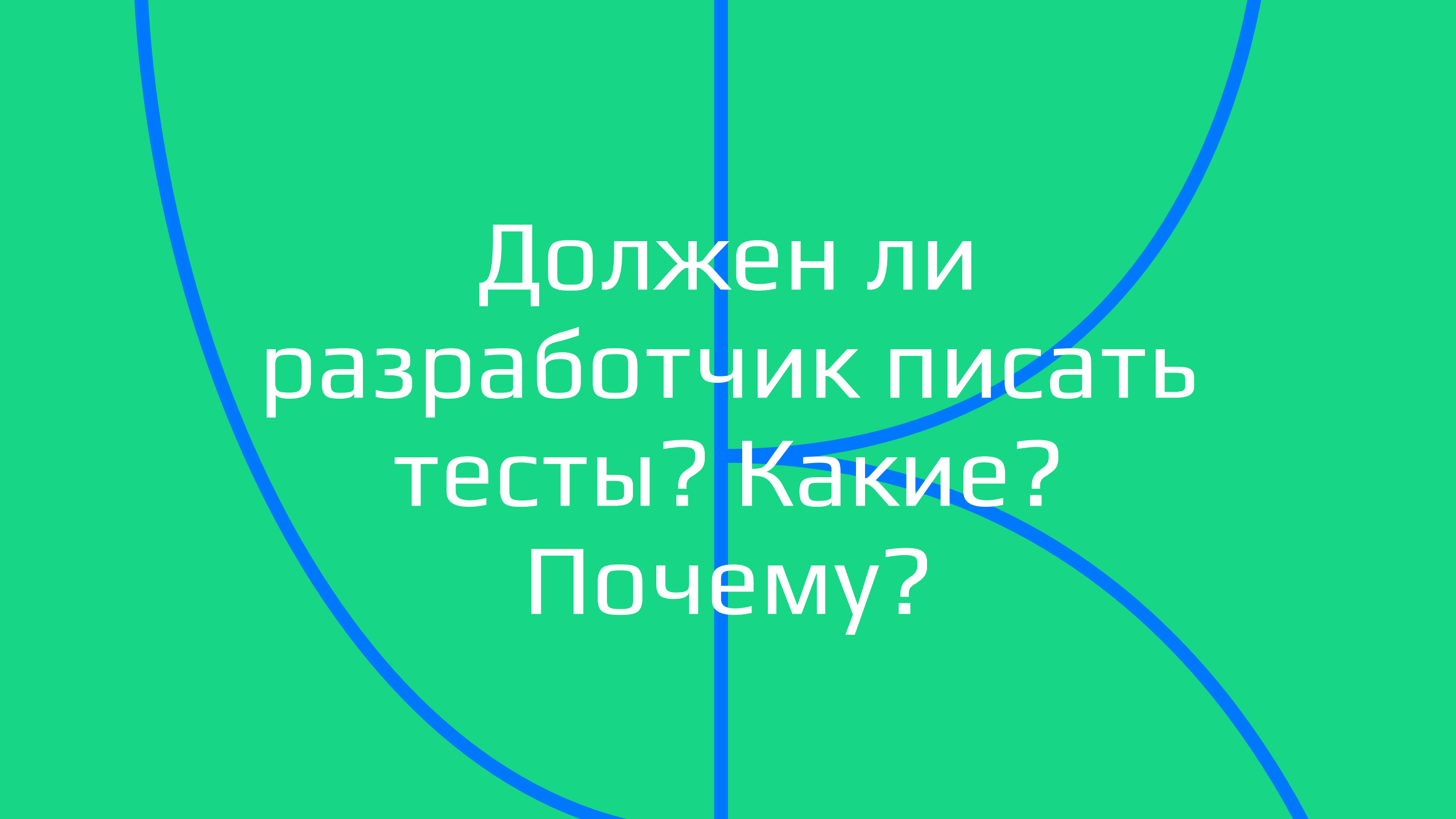
**runtimeOnly** - модуль не виден на этапе компиляции. Но будет добавлен в конечный результат.

и еще несколько утилитарных  
<https://developer.android.com/studio/build/dependencies>

# Автотесты

Зачем и почему?

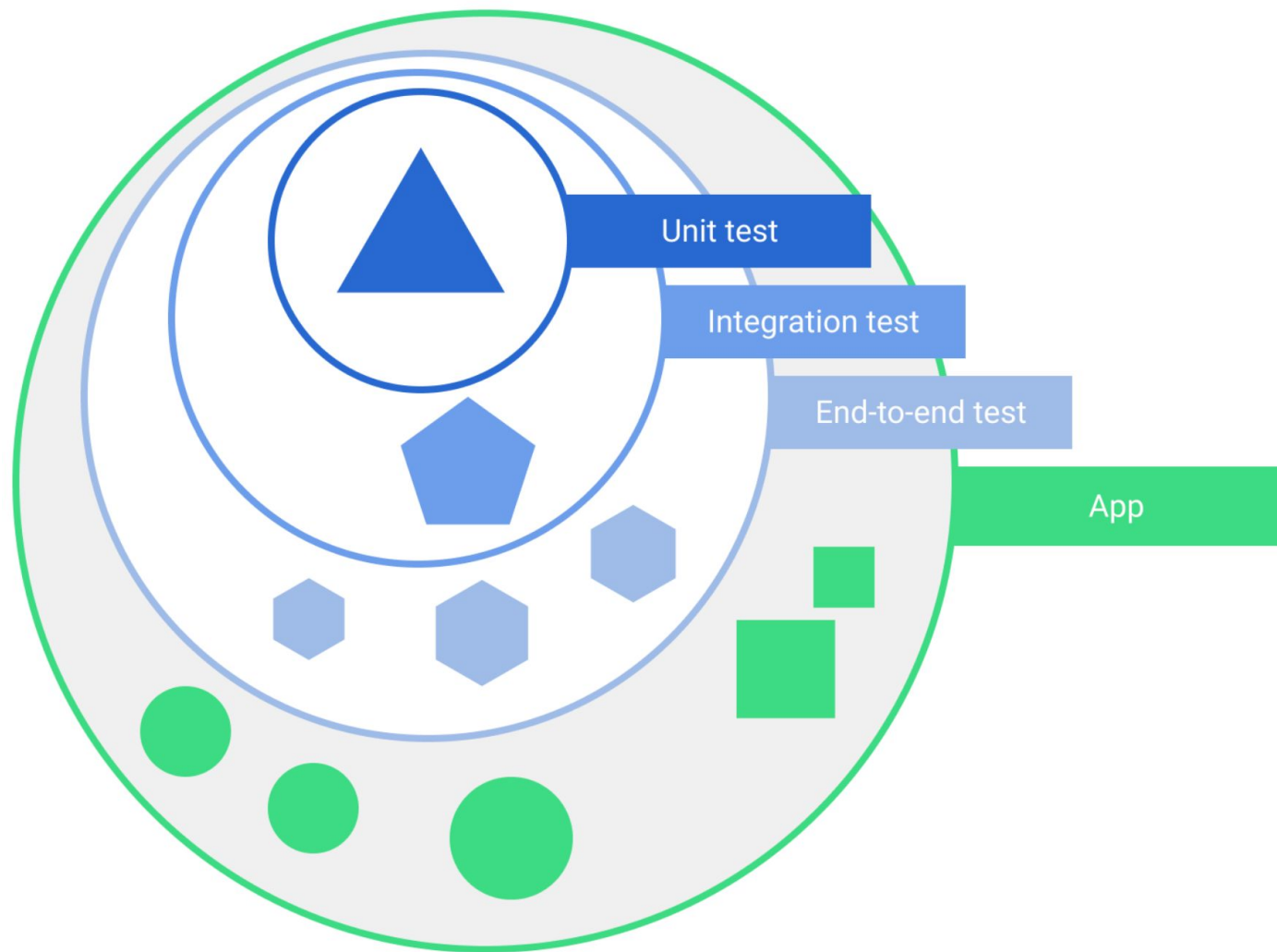




Должен ли  
разработчик писать  
тесты? Какие?  
Почему?

# Классификация тестов

\*одна из...





# Тесты можно поделить на две категории, в зависимости от среды запуска

## Тесты, которые запускаются на jvm

В документации - **local tests**

Видели папочку **test**?:)

По дефолту jvm среда, естественно, не содержит компоненты Android. Поэтому такие тесты желательно писать на компоненты, которые не завязаны на Android.

Если потребуется все же писать тест в такой среде, то Android компоненты тут придется симулировать или мокать и выставлять “дефолтное поведение”.

## Тесты, которые запускаются в реальной среде

В документации - **instrumented test**

Видели папочку **androidTest**?:)

Тест запускается на эмуляторе или на Android. И в своем окружении содержит Android компоненты.

В такой среде можно писать любые виды тестов.

# Build local unit tests

Самый обычный тест можно будет создать и запустить без особых проблем.

Для этого потребуется в папке **<module>/src/test/** создать класс.

К методам этого класса просто прикрепить аннотацию **@Test**

Простой способ запустить тест - это слева от названия этого класса или его методов нажать на зеленый треугольник.

```
// for android objects
android {
    testOptions {
        unitTests.returnDefaultValues = true
    }
}
```

```
// main dependency for test
testImplementation 'junit:junit:4.+'
```

```
class ServiceLocatorTest {
    @Test
    fun stack_test_add() {
        ...
    }
}
```

# Дополнительные зависимости

**Robolectric** - “симуляция” Android, чтобы можно было запускать тесты на JVM.

**Mockito** и **Mockk** - фреймворк для “заглушек”.  
Позволяет вместо реальных объектов делать фейковые объекты, чтобы можно было корректировать поведение или проверять вызовы.

```
dependencies {  
    // Optional -- Robolectric environment  
    testImplementation "androidx.test:core:$..."  
    // Optional -- Mockito framework  
    testImplementation "org.mockito:mockito-core:$..."  
    // Optional -- mockito-kotlin  
    testImplementation "org.mockito.kotlin:mockito-kotlin:$..."  
    // Optional -- Mockk framework  
    testImplementation "io.mockk:mockk:$..."  
}
```

# Спасибо, моки, что вы есть!

Идеальный вариант для Unit-тестирования.

Мы можем создать мок нужного нам объекта и положить его в исследуемый Unit.

Моку можно задать, какой результат должен будет вернуть метод.

А после проведения теста - проверить, какие вызовы были сделаны у мока.

```
val foo = mockk<Foo>(relaxUnitFun = true)
every { foo.data() } returns emptyList()
```

```
val bar = Bar(foo)
bar.load()
```

```
verify { foo.data() }
```

# Для OkHttpClient можно мокать сам запросы

Сам фреймворк называется **MockWebServer**.

<https://github.com/square/okhttp/tree/master/mockwebserver>

Поддерживается той же командой, что делают **OkHttpClient** и **Retrofit**.

Можно настраивать ответы на любые методы.

Было бы такое полезно? Кейсы?

```
public void test() throws Exception {
    MockWebServer server = new MockWebServer();

    // Schedule some responses.
    server.enqueue(new MockResponse().setBody("hello, world!"));
    server.start();

    HttpUrl baseUrl = server.url("/v1/chat/");
    Chat chat = new Chat(baseUrl);
    chat.loadMore();

    assertEquals("hello, world!", chat.messages());

    server.shutdown();
}
```

# Build instrumented tests

Студия для этого вида теста, добавляет необходимые библиотеки и параметры при создании модуля.

В данном случае потребуется аннотировать еще и сам класс раннером, при помощи аннотации **@RunWith**.

```
// main dependencies for instrumented tests
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'

// runner
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"

@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        ...
    }
}
```

# Дополнительные зависимости

**UI Automator** - Аналогия Espresso для проведения UI тестирования.

O! Compose!;)

```
dependencies {  
    // Optional - syntax sugar  
    androidTestImplementation "androidx.test.ext:junit-ktx:..."  
    // Optional -- UI testing with UI Automator  
    androidTestImplementation "androidx.test.uiautomator:uiautomator:..."  
    // Optional -- UI testing with Compose  
    androidTestImplementation "androidx.compose.ui:ui-test-junit4:..."  
    // Optional -- Mocks  
    androidTestImplementation "io.mockk:mockk-android:..."  
}
```

# Espresso?

Основной и базовый фреймворк для написания тестов.

Он делает все необходимые обертки для того, чтобы написание тестов было максимально простым.

Есть много фреймворков, которые в основе содержат Espresso и добавляют сверху еще больше своего сахара, для быстрого написания тестов. Один из таких примеров - **Kakao**.



UI TESTING FOR ANDROID  
**espresso**



# @Before и @After

**@BeforeClass** - исполняет код до всех тестов один раз;

**@Before** - исполняет код перед каждым тестом;

**@After** - исполняет код после каждого теста;

```
@BeforeClass
fun setup() {
    ...
}

@Before
fun prepare() {
    val mockLogger = mockk<Logger.ILogger>(relaxUnitFun = true)

    Logger.initialize(mockLogger)
}

@After
fun clean() {
    ...
}
```

# Минутка рекламы

Понимание как работает Espresso - важно, чтобы прикидывать возможности автотестов.

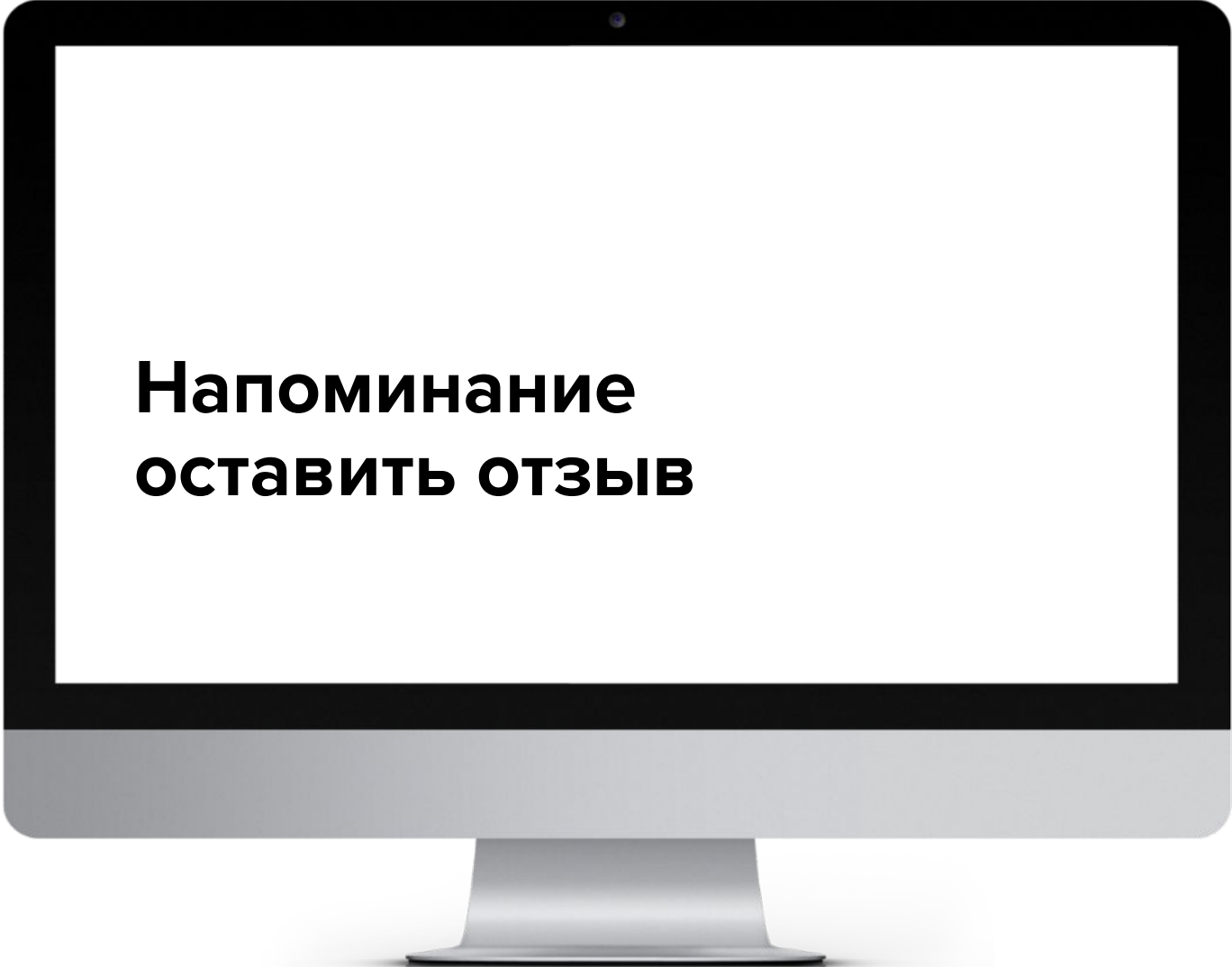
Но я бы рекомендовал бы посмотреть в сторону какого-то более полного фреймворка.

Мои коллеги и бывшие коллеги для автотестов сделали выбор в сторону **Kaspresso**

<https://github.com/KasperskyLab/Kaspresso>

Kaspresso сразу помогает работать с двумя видами фреймворков - Kakao и UiAutomator





**Напоминание  
оставить отзыв**

Спасибо за  
внимание!

