3.12. Тестирование пользовательского интерфейса

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 3.12. Тестирование пользовательского интерфейса

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:52

Оглавление

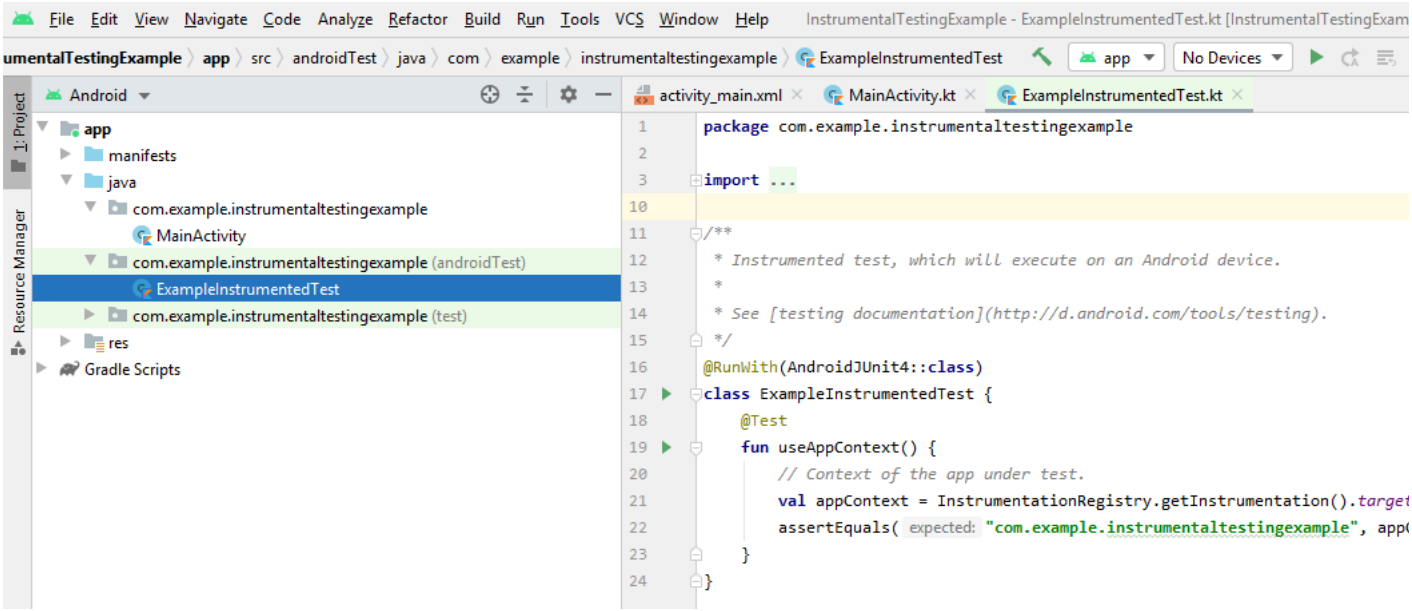
3.12.1 Инструментальные тесты

Инструментальные тесты – это тесты, работающие на реальных устройствах или эмуляторах которые благодаря этому могут использовать все возможности системы Android. Такие тесты нужны в первую очередь для модульных тестов, когда для них требуется использование каких-либо классов Android. Сюда относится тестирование работы с базой данных, с SharedPreferences, с Context и другими классами. Поскольку в нашем курсе пишем Android-приложения, то без классов, которые которые взаимодействуют с классами из Android API не обойтись.

Для инструментальных тестов наличие устройства или эмулятора обязательно. Они позволяют тестировать взаимодействие пользователя с UI (нажатие кнопки, ввод текста, прокрутку, касания и другие операции).

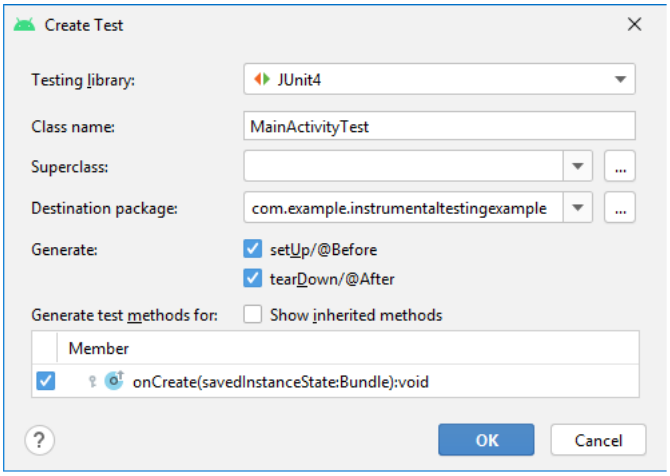
Инструментальные тесты проекта располагаются в папке */src/androidTest/java/*. Поскольку инструментальные тесты располагаются отдельно от вашего приложения APK, они должны иметь свой собственный AndroidManifest.xml файл. Однако Gradle автоматически генерирует этот файл во время сборки, поэтому он не отображается в исходном проекте. При необходимости вы можете добавить свой собственный файл манифеста, например указать другое значение для `minSdkVersion`. При создании приложения Gradle объединяет несколько файлов манифеста в один манифест.

Когда вы создаете новый проект Android Studio создаст пример тестового файла для автоматического тестирования ExampleInstrumentedTest.kt

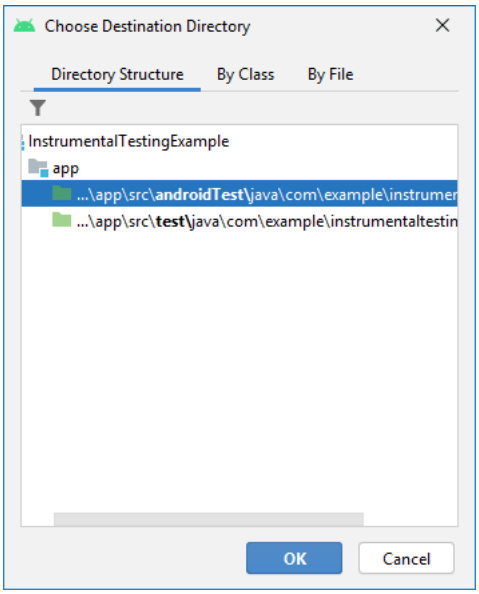


Чтобы создать инструментальный тест подобно локальному unit-тесту, вы можете создать тест для определенного класса или метода, выполнив следующие действия:

1. Откройте файл, содержащий код, который вы хотите протестировать;
2. Выберете класс или метод, который вы хотите протестировать, затем нажмите Ctrl+Shift+T;
3. В появившемся меню нажмите кнопку "Create New Test";
4. Появиться диалоговое окно, в котором нужно выбрать необходимые параметры:



5. Далее нужно выбрать директорию для создания файла с тестами



В итоге на будет создан файл-заготовка для написания инструментальных тестов:

```
import org.junit.After
import org.junit.Before
import org.junit.Test

import org.junit.Assert.*

class MainActivityTest {

    @Before
    fun setUp() {
    }

    @After
    fun tearDown() {
    }

    @Test
    fun onCreate() {
    }
}
```

Кроме этого нужно убедиться, что в сборщик Gradle были добавлены необходимые зависимости. В AndroidX Test доступны следующие зависимости для Gradle:

```
dependencies {
  // Core library
  androidTestImplementation 'androidx.test:core:1.0.0'

  // AndroidJUnitRunner and JUnit Rules
  androidTestImplementation 'androidx.test:runner:1.1.0'
  androidTestImplementation 'androidx.test:rules:1.1.0'

  // Assertions
  androidTestImplementation 'androidx.test.ext:junit:1.0.0'
  androidTestImplementation 'androidx.test.ext:truth:1.0.0'
  androidTestImplementation 'com.google.truth:truth:0.42'

  // Espresso dependencies
  androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.0'
  androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.0'
  androidTestImplementation 'androidx.test.espresso:espresso-intents:3.1.0'
  androidTestImplementation 'androidx.test.espresso:espresso-accessibility:3.1.0'
  androidTestImplementation 'androidx.test.espresso:espresso-web:3.1.0'
  androidTestImplementation 'androidx.test.espresso.idling:idling-concurrent:3.1.0'

  // The following Espresso dependency can be either "implementation"
  // or "androidTestImplementation", depending on whether you want the
  // dependency to appear on your APK's compile classpath or the test APK
  // classpath.
  androidTestImplementation 'androidx.test.espresso:espresso-idling-resource:3.1.0'
}
```

Подробнее о библиотечных зависимостях и дополнительных сведениях можно посмотреть в официальной документации для построения инструментальных модульных тестов.

На сегодняшний день наиболее популярными библиотеками для созданий инструментальных тестов выступают:

- Espresso
- UIAutomator
- Robolectric

У разработчиков есть четкая шпаргалка, на которую можно использовать при разработке тестовых сценариев. В приведенной ниже шпаргалке содержится большинство методов, которые доступны в Espresso. Эта шпаргалка содержит наиболее доступные экземпляры `Matcher`, `ViewAction` и `ViewAssertion`.

```
onView(ViewMatcher)
    .perform(ViewAction)
    .check(ViewAssertion);
```

```
onData(ObjectMatcher
    .DataOptions
    .perform(ViewAct
    .check(ViewAsser
```

View Matchers

USER PROPERTIES
```
withId(...)
withText(...)
withTagKey(...)
withTagValue(...)
hasContentDescription(...)
withContentDescription(...)
withHint(...)
withSpinnerText(...)
hasLinks()
hasEllipsizedText()
hasMultilineTest()
```

HIERARCHY
```
withParent(Matcher)
withChild(Matcher)
hasDescendant(Matcher)
isDescendantOfA(Matcher)
hasSibling(Matcher)
isRoot()
```

INPUT
```
supportsInputMethods(...)
hasIMEAction(...)
```

UI PROPERTIES
```
isDisplayed()
isCompletelyDisplayed()
isEnabled()
hasFocus()
isClickable()
isChecked()
isNotChecked()
withEffectiveVisibility(...)
isSelected()
```

CLASS
```
isAssignableFrom(...)
withClassName(...)
```

ROOT MATCHERS
```
isFocusable()
isTouchable()
isDialog()
withDecorView()
isPlatformPopup()
```

OBJECT MATCHER
```
allOf(Matchers)
anyOf(Matchers)
is(...)
not(...)
endsWith(String)
startsWith(String)
instanceOf(Class)
```

SEE ALSO
```
Preference matchers
Cursor matchers
Layout matchers
```

Data Options

```
inAdapterView(Matcher)
atPosition(Integer)
onChildView(Matcher)
```

View Actions

CLICK/PRESS
```
click()
doubleClick()
longClick()
pressBack()
pressIMEActionButton()
pressKey([int/EspressoKey])
pressMenuKey()
closeSoftKeyboard()
openLink()
```

GES
```
scro
swip
swip
swip
swip
```

TEX
```
clea
type
type
repl
```

View Assertio

```
matches(Matcher)
doesNotExist()
selectedDescendantsMatch(...)
```

LAYOUT ASSERTIONS
```
noEllipseizedText(Matcher)
noMultilineButtons()
noOverlaps([Matcher])
```

POS
```
isLe
isRi
isLe
isRi
isAb
isBe
isBo
isTo
```

```
intended(IntentMatcher);
```

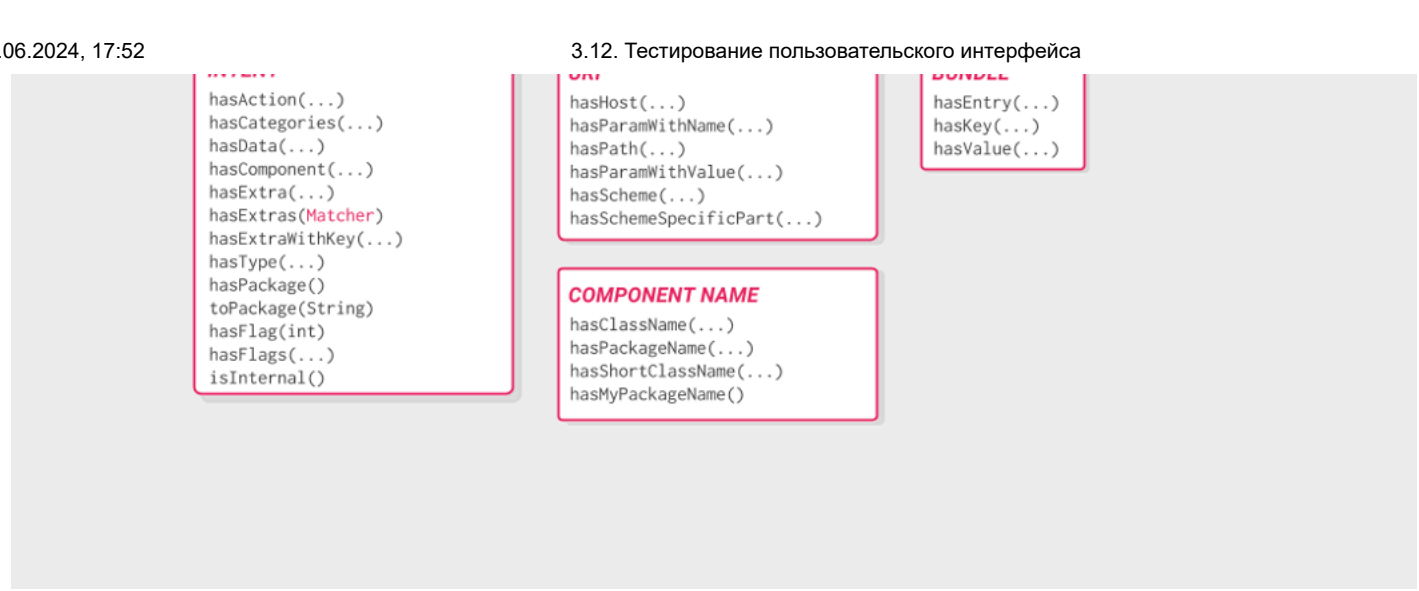
```
intending(IntentMatch
    .respondWith(Act
```

Intent Matchers

INTENT

URI

BUNDLE



В данной теме познакомимся с фреймворком Espresso. Далее продемонстрируем некоторые его основные возможности.

3.12.2. Введение в Espresso для Android

Espresso - это простой, быстрый и настраиваемый фреймворк тестирования Android-приложений, который предназначен для разработки надежных автоматизированных тестов пользовательского интерфейса. На сегодняшний день он является одним из самых популярных тестовых фреймворков для платформы Android.

Библиотека Espresso, предоставляемая AndroidX Test, предоставляет API-интерфейсы для написания UI-тестов имитации взаимодействия с пользователем. Тесты Espressoможно запускать на устройствах с Android 4.0.1 (уровень API 14) и выше. Ключевым преимуществом использования библиотеки Espresso является то, что она обеспечивает автоматическую синхронизацию тестовых действий с пользовательским интерфейсом приложения.

Для создания тестов с помощью фреймворка Espresso в gradle необходимо присутствие следующей зависимости:

```
dependencies {
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.0'
}
```

Перед тем как перейти к непосредственно к возможностям библиотеки Espresso, давайте подумаем о мобильном приложении с точки зрения конечного пользователя. Что делают пользователи, когда они используют приложение? Сначала они ищут необходимые элементы UI на экране приложения (кнопки, списки, текстовые поля и т.д.), затем выполняют некоторые действия с ними (нажимают кнопки, вводят текст и т.д.), далее проверяют результат работы (привело ли нажатие кнопки или свайп к желаемому результату, введен ли текст и т.д.).

Таким образом при инструментальном тестировании возникают следующие задачи:

- поиск элементов пользовательского интерфейса в приложении;
- проверка действий над элементами UI;
- проверка результатов действий над элементами пользовательского интерфейса.

Для решения этих задач в Espresso есть три типа методов :

- `ViewMatchers` — позволяют найти объект в текущей иерархии представлений
- `ViewAssertions` — позволяют проверить состояние объекта и подтвердить, что состояние соответствует критериям
- `ViewActions` — эти методы позволяют выполнять различные действия с объектами.

Таким образом, общая схема всех тестов с Espresso выглядит так:

1. Найти View, передав в метод onView объект Matcher.
2. Выполнить какие-то действия над этой View, передав в метод perform объект ViewAction.
3. Проверить состояние View, передав в метод check объект ViewAssertion.

Обычно для создания объекта ViewAssertion используют метод matches, который принимает объект Matcher.

Espresso – это очень умный фреймворк, который грамотно проверяет все элементы, при этом он может ждать некоторое время, пока выполнится определенное условие, что очень удобно, так как не всегда элементы на экране появляются мгновенно. Для каждого из этих объектов существует большое количество стандартных методов, которые позволяют покрыть подавляющее большинство сценариев проверки:

1. withId, withText, withHint, withTagKey, ... – Matcher.
2. click, doubleClick, scrollTo, swipeLeft, typeText, ... – ViewAction.
3. matches, doesNotExist, isLeftOf, noMultilineButtons – ViewAssertion.

Если вам не хватит стандартных объектов, вы всегда можете создать свои собственные. Далее подробнее рассмотрим ключевые методы библиотеки в Kotlin.

3.12.3 Основы работы с Espresso в Kotlin

В данном разделе показано, как выполнять стандартные задачи автоматического тестирования с помощью Espresso API в Kotlin. Основные компоненты Espresso следующие:

- `Espresso` - точка входа для взаимодействия с представлениями (осуществляется с помощью методов `onView()` и `onData()`).
- `ViewMatchers` - коллекция, реализующих интерфейс `Matcher<? super View>`. Для поиска представлений в текущей иерархии, можно передать один или несколько объектов коллекции методу `onView()`.
- `ViewActions` - коллекция объектов `ViewAction`, которые можно передать методу `ViewInteraction.perform()`, например метод `click()`. Позволяет взаимодействовать с компонентами (click, longClick, doubleClick, swipe, scroll и т.д.).
- `ViewAssertions` - объекты `ViewAssertion` можно передать методу `ViewInteraction.check()`. Чаще всего используется для подтверждения состояния представления.

Приведем пример:

```
// withId(R.id.my_view) объект ViewMatcher
// click() объект ViewAction
// matches(isDisplayed()) объект ViewAssertion
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()))
```

Найти представление по его id можно следующим образом:

```
onView(withId(R.id.my_view))
```

Кроме того можно найти представление по размещенному в нем тексту. Например найти представление с текстом "Hello" можно следующим образом:

```
onView(allOf(withId(R.id.my_view), withText("Hello!")))
```

Можно поставить обратное условие

```
onView(allOf(withId(R.id.my_view), not(withText("Hello!"))))
```

Более подробно о возможностях `ViewMatchers` можно посмотреть в официальной документации

После того как необходимое представление найдено в иерархии, можно выполнить для него метод perform класса ViewAction. В нем мы указываем конкретное действие, которое нужно выполнить. Приведем некоторые методы:

- `ViewActions.click()` - возвращает действие нажатия кнопки;
- `ViewActions.typeText()` - возвращает действие ввода текста;
- `ViewActions.pressKey()` - возвращает действие нажатия на клавишу;
- `ViewActions.clearText()` - возвращает действие, очищающее текст в представлении
- и т.д.

Подробнее с методами класса можно ознакомиться в официальной документации.

Например, для клика по представлению в тесте необходимо написать следующее:

```
onView(...).perform(click())
```

Вы можете выполнить более одного действия за один вызов функции:

```
onView(...).perform(typeText("Hello"), click())
```

Методы класса `ViewActions` могут применяться к текущему выбранному представлению с помощью метода `check()`. При этом наиболее часто используемый метод - метод `matches()`. Например, чтобы проверить, есть ли в представлении текст «Hello!» нужно написать следующее:

```
onView(...).check(matches(withText("Hello!")))
```

Возможности библиотеки довольно обширны. Также доступно тестирование адаптеров, потоков и др. Далее рассмотрим два простых примера разработки инструментальных тестов и использованием Espresso.

3.12.4 Пример UI тестирования в Espresso

Разберем простой пример тестирования приложения в Espresso, имитирующего процесс авторизации. Приложение будет состоять из двух активностей. На главной активности будет располагаться кнопка "Login" по нажатию которой запускается активность LoginActivity с формой для авторизации.

Для начало в gradle проекта необходимо добавить (проверить наличие) следующие зависимости:

```
android {
    //...
    defaultConfig {
        //...
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    //...
}

dependencies {
    //...
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test:rules:1.0.2'

}
```

Файл разметки *activity_ main.xml* будет выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout 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"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/btn_login"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Login"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:onClick="login"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Файл разметки окна авторизации *activity_ login.xml* представлен ниже:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout 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"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".LoginActivity">

    <TextView
        android:id="@+id/tv_login"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Login"
        android:textSize="30sp" />

    <EditText
        android:id="@+id/et_username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="username"/>

    <EditText
        android:id="@+id/et_password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="password"
        android:inputType="textPassword" />

    <Button
        android:id="@+id/btn_submit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Submit" />
</LinearLayout>
```

В файле MainActivity.kt напишем метод обработки нажатия кнопки. При нажатии этой кнопки открывается экран LoginActivity. Этот экран содержит два EditText (поля имени пользователя и пароля) и кнопку «Submit».

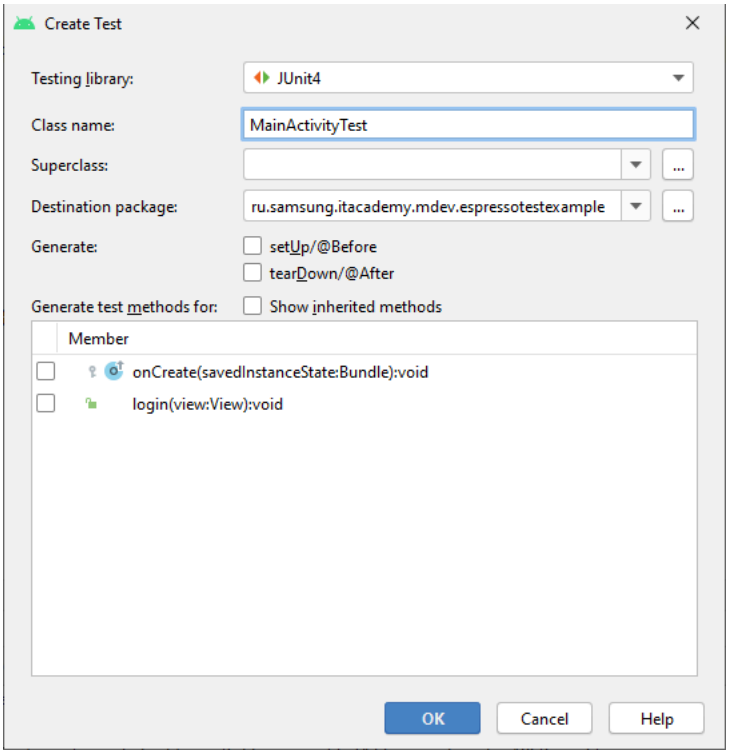
```
package ru.samsung.itacademy.mdev.espressotestexample

import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View

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

    fun login(view: View) {
        val changePage = Intent(this, LoginActivity::class.java)
        startActivity(changePage)
    }
}
```

Теперь напишем тест для класса MainActivity. Перейдем в класс MainActivity, поместим курсор к имени MainActivity и нажмем комбинацию клавиш *Ctrl + Shift +T*. Далее во всплывающем меню выберем *"Create New Test"*.



Нажмите кнопку OK, и появится другое диалоговое окно. Выберите каталог androidTest и еще раз нажмите кнопку OK. Обратите внимание: поскольку мы пишем инструментальный тест (специальные тесты для Android SDK), тестовые примеры находятся в папке androidTest/java.

Android Studio успешно создала для нас тестовый класс. Над именем класса включим аннотацию: `@RunWith (AndroidJUnit4 :: class)`, а также импортируем необходимые библиотеки. Аннотация означает, что все тесты в этом классе являются тестами для Android. В итоге класс для тестов будет выглядеть следующим образом:

```
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class MainActivityTest{

}
```

Поскольку мы хотим протестировать Activity, предварительно нужно сообщить Espresso, какую активность следует тестировать. Для этого в класс добавим следующий код:

```
@Rule @JvmField
var activityRule = ActivityTestRule<MainActivity>(MainActivity::class.java)
```

Аннотация @Rule означает, что это тестовое правило библиотеки JUnit4. Правила тестирования JUnit4 запускаются до и после каждого метода тестирования, помеченного аннотацией @Test. В нашем сценарии мы хотим запускать MainActivity перед каждым методом тестирования и уничтожать ее после его выполнения.

Мы также добавили аннотацию @JvmField, которая указывает компилятору не создавать геттеры и сеттеры для activityRule, а вместо этого предоставлять его как простое поле Java.

Напомним три ключевых этапа написания тестов в Espresso:

1. поиск виджета (например, TextView или Button), который хотите протестировать;
2. выполнение одного или нескольких действий с этим виджетом;
3. проверка состояния виджета.

В файле разметки MainActivity у нас есть только один виджет - кнопка входа в систему. Давайте протестируем сценарий, в котором пользователь найдет эту кнопку и нажмет на нее. Напишем внутри тестового класса следующий код:

```
@Test
@Throws(Exception::class)
fun clickLoginButton() {
    onView(withId(R.id.btn_login))
}
```

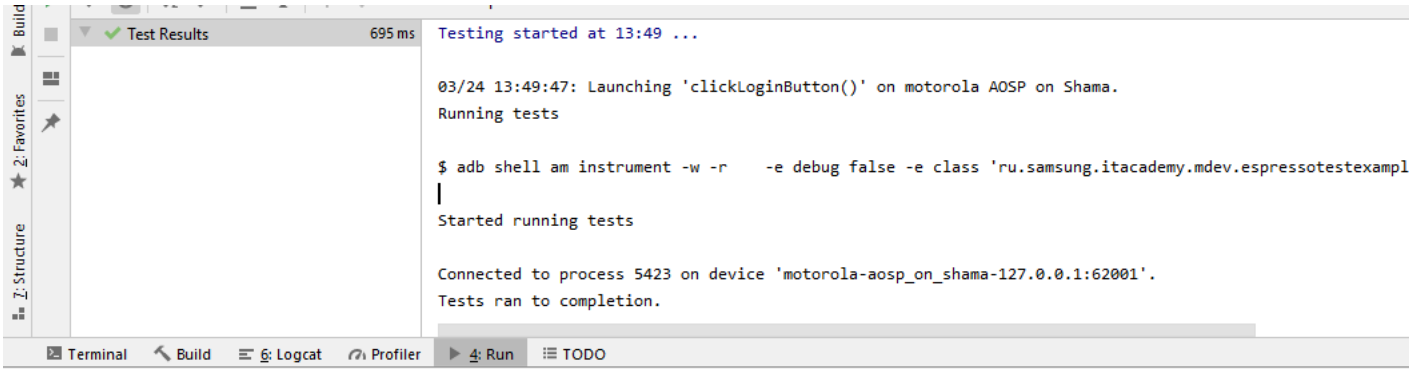
Чтобы найти виджеты в Espresso, мы используем статический метод `onView()`. Тип параметра, который мы предоставляем `onView()` - это `Matcher`. Обратите внимание, что `Matcher` необходимо импортировать из проекта Hamcrest. Метод `onView(withId(R.id.btn_login))` вернет объект `ViewInteraction`, который предназначен для виджета с идентификатором `R.id.btn_login`. В приведенном примере мы использовали `withId()` для поиска виджета с заданным идентификатором. Кроме этого можно использовать другие методы (`withText()`, `withHint()` и т.д.).

Для начала проверим, действительно ли кнопка отображается на экране. Для этого в метод `clickLoginButton()` добавим строчку:

```
onView(withId(R.id.btn_login)).check(matches(isDisplayed()))
```

Здесь мы просто проверяем, видна ли кнопка с данным идентификатором для пользователя, поэтому мы используем метод `check()`, чтобы подтвердить, имеет ли представление определенное состояние - в нашем случае, если оно видимо на экране. Статический метод `matches()` возвращает объект `ViewAssertion`, который говорит о том, что представление существует в иерархии представлений и соответствует данному сопоставлению.

Для запуска теста можно нажать на зеленый треугольник рядом с методом или именем класса. Щелчок по треугольнику рядом с именем класса запустит все методы тестирования в этом классе, а по треугольнику рядом с методом, запустит тест только для этого метода. После нажатия видим, что тест выполнился.



В объекте `ViewInteraction`, который возвращается вызовом метода `onView()`, мы можем имитировать действия, которые пользователь может выполнить с виджетом. Например, мы можем имитировать нажатия, вызвав статический метод `click()`. Добавим в метод `clickLoginButton()` следующую строчку кода:

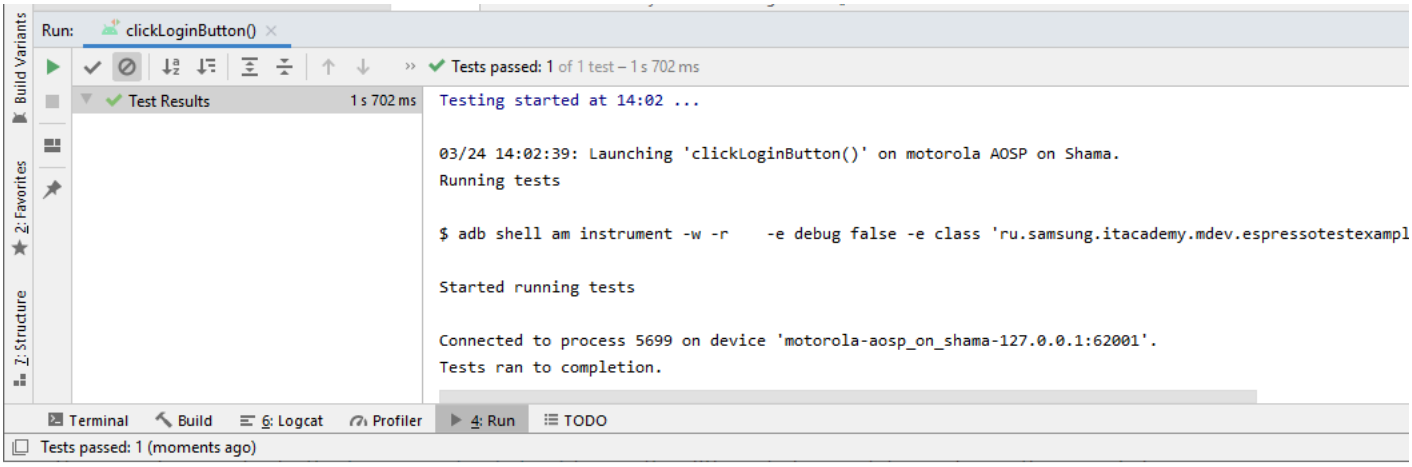
```
onView(withId(R.id.btn_login)).perform(click())
```

Для имитации нажатия сначала вызывается метод `perform()`. Он выполняет указанное действие(я) над виджетом. Обратите внимание, что можно передать методу несколько действий через запятую. В данном примере мы передали единственное действие `click()`. О том какие еще действия можно тестировать говорилось в предыдущем разделе данной темы. С ними можно ознакомиться в официальной документации.

Давайте завершим наш тест, убедившись в том, что после нажатия кнопки открылась нужная нам активность. Для этого в метод теста можно добавить следующую строчку кода:

```
onView(withId(R.id.tv_login)).check(matches(isDisplayed()))
```

Внутри файла разметки LoginActivity у нас также есть TextView с идентификатором R.id.tv_login. Поэтому здесь просто проверяем, что TextView с заданным id виден пользователю. Запустим тест и убедимся в его выполнении:



Теперь напишем тесты для LoginActivity. Сначала в файл LoginActivity.kt добавим следующий код:

```
class LoginActivity : AppCompatActivity() {
    private lateinit var usernameEditText: EditText
    private lateinit var loginTitleTextView: TextView
    private lateinit var passwordEditText: EditText
    private lateinit var submitButton: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)

        usernameEditText = findViewById(R.id.et_username)
        passwordEditText = findViewById(R.id.et_password)
        submitButton = findViewById(R.id.btn_submit)
        loginTitleTextView = findViewById(R.id.tv_login)

        submitButton.setOnClickListener {
            if (usernameEditText.text.toString() == "emityakov" &&
                passwordEditText.text.toString() == "password") {
                loginTitleTextView.text = "Success"
            } else {
                loginTitleTextView.text = "Failure"
            }
        }
    }
}
```

В приведенном коде, если введенное имя пользователя - «emityakov», а пароль - «password», то вход в систему будет выполнен успешно. Для любого другого входа это произойдет сбой. Давайте напишем тест для такого сценария.

Создадим класс LoginActivityTest.kt подобно тому как мы уже делали с MainActivity. Код класса будет выглядеть следующим образом:

```
package ru.samsung.itacademy.mdev.espressotestexample

import androidx.test.espresso.Espresso
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.espresso.matcher.ViewMatchers.withText
import androidx.test.rule.ActivityTestRule
import org.junit.Assert.*
import org.junit.Rule
import org.junit.Test

class LoginActivityTest{
    @Rule
    @JvmField
    var activityRule = ActivityTestRule<LoginActivity>(
        LoginActivity::class.java
    )

    private val username = "emityakov"
    private val password = "password"

    @Test
    fun clickLoginButton() {
        onView(withId(R.id.et_username)).perform(ViewActions.typeText(username))
        onView(withId(R.id.et_password)).perform(ViewActions.typeText(password))

        onView(withId(R.id.btn_submit)).perform(ViewActions.click())

        Espresso.onView(withId(R.id.tv_login))
            .check(matches(withText("Success")))
    }

}
```

Этот тестовый класс очень похож на предыдущий. Если мы запустим тест, откроется наш экран LoginActivity. Имя пользователя и пароль вводятся в поля с идентификаторами R.id.et_username и R.id.et_password соответственно. Затем Espresso нажмет кнопку «Submit» (R.id.btn_submit) и будет ждать, пока не будет найдено представление с идентификатором R.id.tv_login с текстом «Успешно».

Код проекта можно посмотреть здесь.

Начать тур для пользователя на этой странице