

# Espresso Workshop Companion

Exercises and additional info

## Table of Contents

Info: Terminology .....	3
Activity .....	3
View.....	3
Exercise: Dismiss the Onboarding Screen .....	5
Info: Where do the IDs come from? .....	7
Info: Espresso Core API - Cheat Sheet.....	8
Exercise: Select, Click and Scroll .....	9
Info: RecyclerViewActions .....	10
Exercise: Don't Waste Time: Recycle with Espresso .....	11
Adding the Sauce .....	12
Exercise: Breaking the Rules.....	13
Bonus Exercise: Intended Intents.....	14
The app (features) .....	16

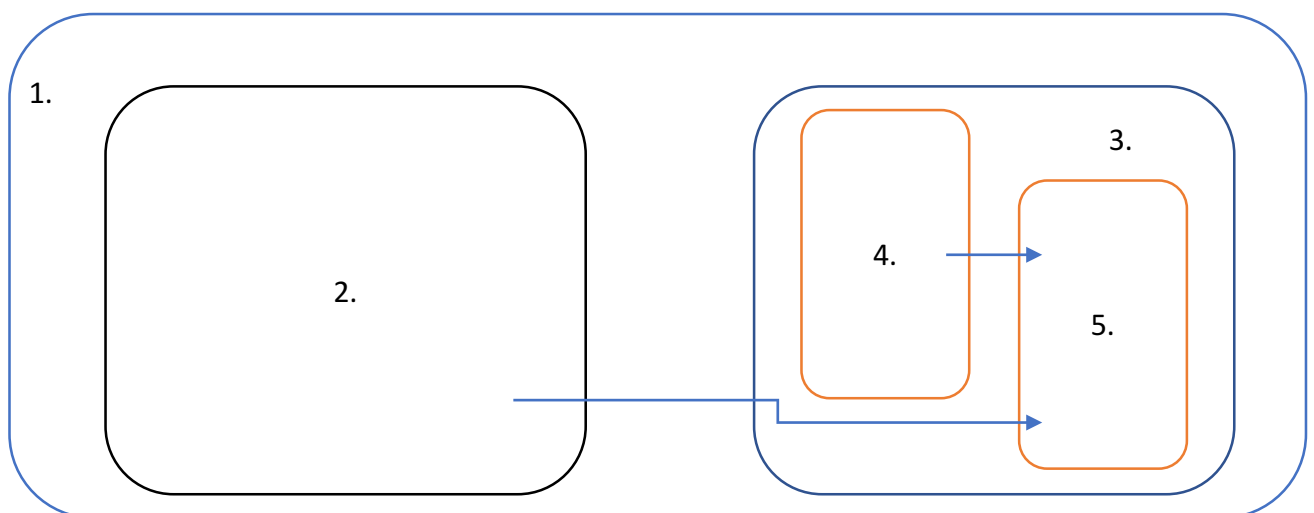
## Info: Terminology

### Activity

The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place. Instead, the user journey often begins non-deterministically. For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

The Activity class is designed to facilitate this paradigm. When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. In this way, the activity serves as the entry point for an app's interaction with the user.

Source: <https://developer.android.com/guide/components/activities.html>



1. Android OS
2. App X  
Has a button that takes you to the 'compose email screen' in your mail app
3. Mail app
4. 'MainActivity' of the App. If you start the app, this is the activity that automatically gets started. It loads your inbox and has a few buttons, for example a 'compose email' button.
5. 'ComposeEmailActivity'. This activity allows you to compose an email. If you press the 'compose email' button in the MainActivity, this gets started. It's also directly accessible by activities from other apps. This means that this activity can also be invoked from App/Activity X in this example.

### View

To simplify things a bit: everything you see in the UI is basically a View. A View occupies a rectangular area on the screen and can be anything such as a Textview, Button, EditText, ImageView, ViewGroup and more. Views are arranged (how and where they are displayed on screen) in a layout. A layout (a form of ViewGroup) is an invisible container that holds

other Views and defines their layout properties.

In short:

a View can be a smaller part of the UI you interact with, such as a button or text field, but a View can also be an invisible container that holds other Views and defines their layout properties.

source: <https://developer.android.com/reference/android/view/View>

## Exercise: Dismiss the Onboarding Screen

### Goal

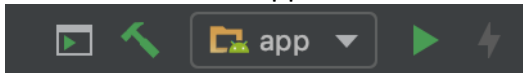
Learn how to use the 'Layout Inspector' to inspect the app UI and find IDs for Views.

### What

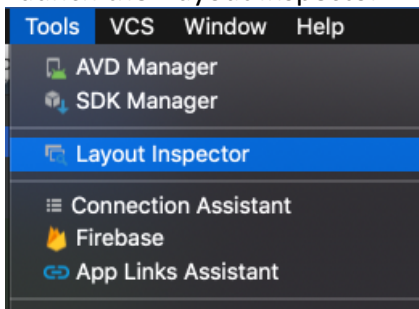
You've been given a test that is broken: the IDs 'example\_id' and 'another\_example\_id' are non-existing.

### How

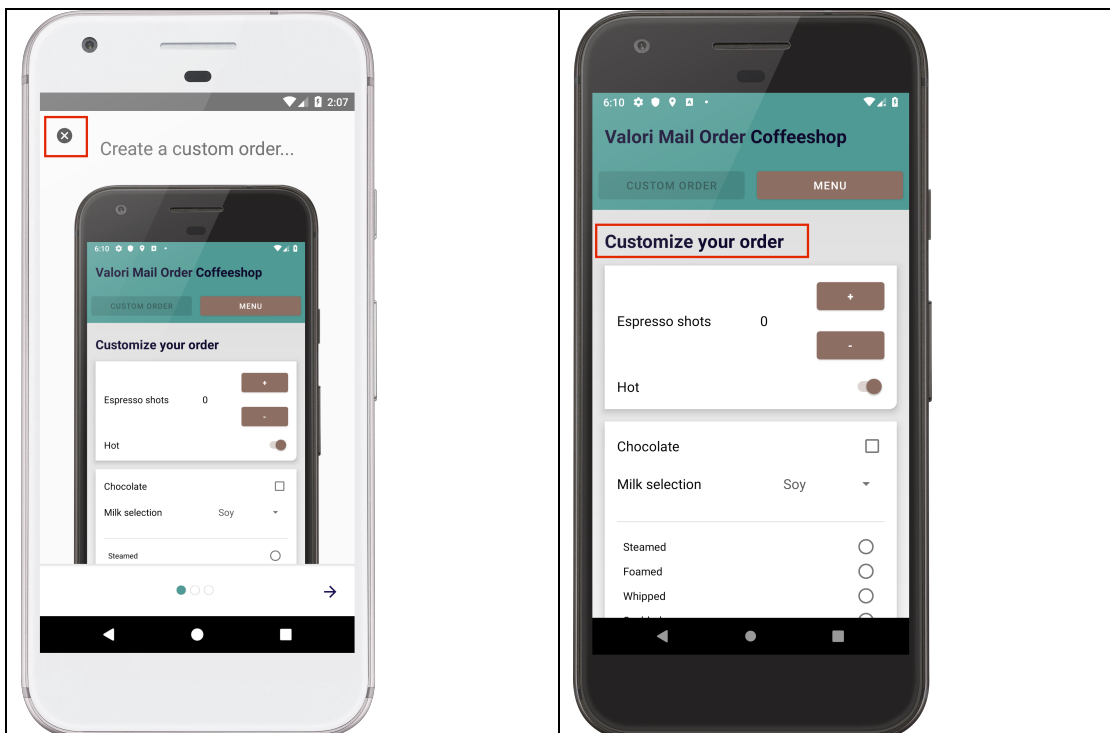
1. Launch the app, by pressing the 'play' button in the top right corner of Android Studio. Make sure 'app' is selected in the drop down menu



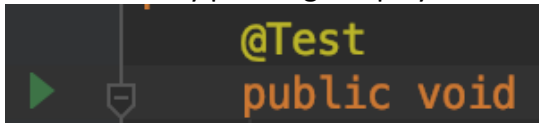
2. Use the app to navigate to the screen you'd like to inspect
3. Launch the 'Layout Inspector'



4. Select the device and activity you'd like to inspect
5. Find the IDs for the following Views:



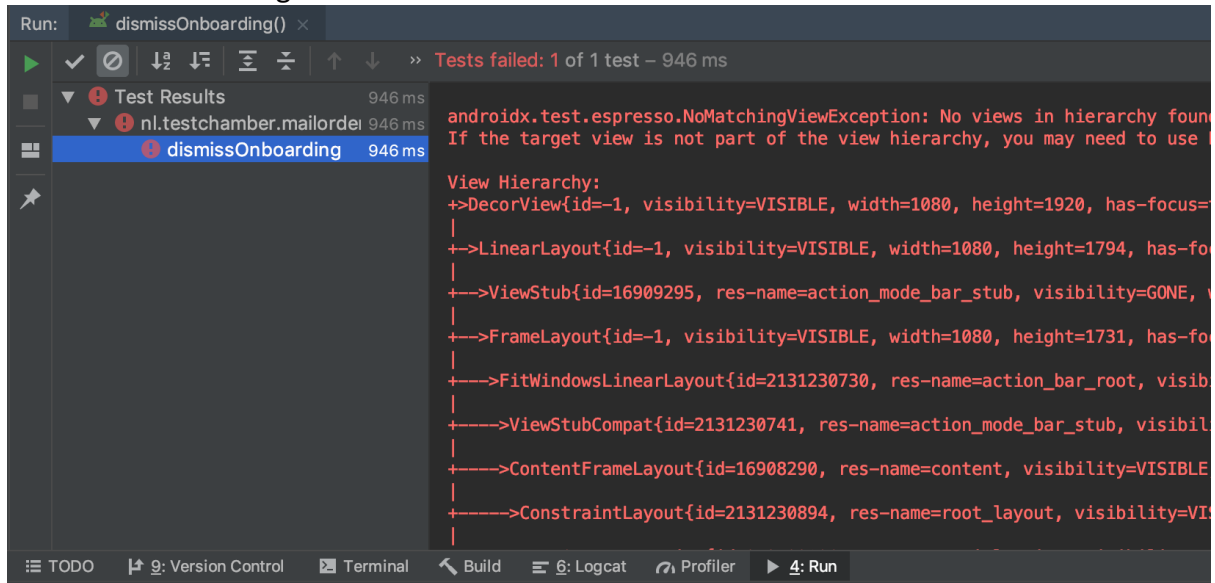
6. Replace the IDs in the test. The proper format is 'R.id.id\_name'.
7. Run the test by pressing the play icon next to the method name:



Or use the ctrl+shift+r short cut.

### Troubleshooting

You can view the result of the test run in the bottom left corner. If the test failed, you should see something like this:



If you select the failed test on the left, you'll be able to see some more detail on the right. A 'NoMatchingViewException' means that the View you were looking for was not found. This means that you've used the wrong View ID, or you were looking for the View at the wrong time (i.e. you were looking for an ID from the onboarding screen, while the app was already on the custom order screen).

## Info: Where do the IDs come from?

View Layouts can be created in XML. In the XML layout you have the option to assign IDs to a View. This can come in handy when arranging the Views. For example, you can state that View with ID A should be displayed beneath the View with ID B. You can also use this ID in your code to assign behaviour or properties to a View (such as changing the text for the View with ID A). The IDs that you create are automatically added to the app Resources. This enables us to use these IDs in the whole project, including our tests.

That's why we use "R.id.\*" in our tests.

If you hold the CMD button (Alt for windows?) and click on the ID, you will be taken to the XML layout file where the ID is defined.

Similarly to IDs, it's good practice to add text that's used in an app to the String Resources. These can be accessed by using "R.string.\*". You shouldn't use these resources for text verification though, because that means you're using the same text that is used to set the text for a View as your expected value: a test that will never fail.

# Info: Espresso Core API - Cheat Sheet

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

**View Matchers**

**USER PROPERTIES**  
withId(...)  
withText(...)  
withTagKey(...)  
withTagValue(...)  
hasContentDescription(...)  
withContentDescription(...)  
withHint(...)  
withSpinnerText(...)  
hasLinks()  
hasEllipsizedText()  
hasMultilineText()

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

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

**HIERARCHY**  
withParent(Matcher)  
withChild(Matcher)  
hasDescendant(Matcher)  
isDescendantOf(Matcher)  
hasSibling(Matcher)  
isRoot()

**INPUT**  
supportsInputMethods(...)  
hasIMEAction(...)

**CLASS**  
isAssignableFrom(...)  
withClassName(...)

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

**SEE ALSO**  
Preference matchers  
Cursor matchers  
Layout matchers

**onData(**ObjectMatcher**)**  
.perform(**ViewAction**)  
.check(**ViewAssertion**);

**Data Options**  
inAdapterView(Matcher)  
atPosition(Integer)  
onChildView(Matcher)

**View Actions**

**CLICK/PRESS**  
click()  
doubleClick()  
longClick()  
pressBack()  
pressIMEActionButton()  
pressKey(Int/EspressoKey)  
pressMenuKey()  
closeSoftKeyboard()  
openLink()

**GESTURES**  
scrollTo()  
swipeLeft()  
swipeRight()  
swipeUp()  
swipeDown()

**TEXT**  
clearText()  
typeText(String)  
typeTextIntoFocusedView(String)  
replaceText(String)

**View Assertions**

**MATCHES**  
matches(Matcher)  
doesNotExist()  
selectedDescendantsMatch(...)

**LAYOUT ASSERTIONS**  
noEllipsizedText(Matcher)  
noMultilineButtons()  
noOverlaps(Matcher)

**POSITION ASSERTIONS**  
isLeftOf(Matcher)  
isRightOf(Matcher)  
isLeftAlignedWith(Matcher)  
isRightAlignedWith(Matcher)  
isAbove(Matcher)  
isBelow(Matcher)  
isBottomAlignedWith(Matcher)  
isTopAlignedWith(Matcher)

**intended(**IntentMatcher**);**

**Intent Matchers**


**INTENT**  
hasAction(...)  
hasCategories(...)  
hasData(...)  
hasComponent(...)  
hasExtra(...)  
hasExtras(Matcher)  
hasExtraWithKey(...)  
hasType(...)  
hasPackage()  
toPackage(String)  
hasFlag(int)  
hasFlags(...)  
isInternal()

**URI**  
hasHost(...)  
hasParamWithName(...)  
hasPath(...)  
hasParamWithValue(...)  
hasScheme(...)  
hasSchemeSpecificPart(...)

**BUNDLE**  
hasEntry(...)  
hasKey(...)  
hasValue(...)

**COMPONENT NAME**  
hasClassName(...)  
hasPackageName(...)  
hasShortClassName(...)  
hasHyPackageName()

**intending(**IntentMatcher**)**  
.respondWith(**ActivityResult**);



**espresso** CHEAT SHEET **2.1**

v2.1.0, 4/21/2015

source: <https://developer.android.com/training/testing/espresso/cheat-sheet>

8



## Exercise: Select, Click and Scroll

### *Goal*

Learn to use the Espresso Core API to write a complete test.

Practice with using the Layout Inspector

Think about (be mindful of) code duplication

### *What*

Write a test where you create a custom order with 2 shots of espresso and add Chocolate.

Verify if the ingredients list on the Order Overview screen contains the right items.

### *How*

Write your own test method. Don't forget to add the `@Test` annotation.

Use the `onView()` method to find a View and chain the `.perform()` and/or `.check()` method to perform an action or create an assertion.

### *Extra*

- If your device is large enough to display the whole screen at once, you don't have to scroll. If you'd like to try it anyway, you can change the device orientation to landscape.

Use the following code:

```
activityTestRule.getActivity().setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

- Feel free to try to write additional tests using other elements from the Custom Order screen.

## Info: RecyclerViewActions

```
actionOnHolderItem(VIEW_HOLDER_MATCHER, VIEW_ACTION)
actionOnItem(VIEW_MATCHER, VIEW_ACTION)
actionOnItemAtPosition(int, VIEW_ACTION)
scrollTo(VIEW_MATCHER)
scrollToHolder(VIEW_HOLDER_MATCHER)
scrollToPosition(int)
```

source:

<https://developer.android.com/reference/android/support/test/espresso/contrib/RecyclerViewActions>

### *Tip*

If you're looking to match based upon certain properties that are available in the UI (such as ID, Text etc.), you should use a RecyclerViewAction that accept a ViewMatcher as an argument. You can use the same ViewMatchers as you would when trying to find a View with the onView() method.

When constructing the ViewMatcher keep in mind that the RecyclerViewActions that accept ViewMatchers, expect you to provide a ViewMatcher that helps it find an *item*. That means it's looking for the root View of the Layout. You can't just provide a matcher that directly matches a nested TextView that is part of the item. You'll have to specify its relation to the root View of the item, from the perspective of the root.

## Exercise: Don't Waste Time: Recycle with Espresso

### *Goal*

Learn to use RecyclerViewActions

Learn how to construct a matcher for RecyclerView items

### *What*

Write a test where you select the "Con Panna" beverage from the menu. Verify the beverage name on the Order Overview screen.

### *How*

Navigate to the menu and use a RecyclerViewAction to click on the "Con Panna" item. Here's a RecyclerViewAction example:

```
onView(withId(R.id.recycler_view_id)).perform(RecyclerViewActions.<RecyclerView.ViewHolder>actionOnItem(VIEW_MATCHER, VIEW_ACTION));
```

Replace the ID in the onView matcher with the ID of the RecyclerView in the app. Pick a RecyclerViewAction and supply the right ViewMatcher to select the "Con Panna" item and pick an action to perform on that item.

### *Extra*

- Feel free to try to write additional tests. For example: enter text into the edit text fields in the Order Overview.

## Adding the Sauce

to do

## Exercise: Breaking the Rules

### *Goal*

Learn about the `ActivityTestRule`

See the possibilities offered by Espresso to use app code

### *What*

Modify the app state before it's launched by setting 'first\_launch' to false in the app's `SharedPreferences`.

### *How*

Override the `beforeActivityLaunched` method and implement the code to edit the `SharedPreferences`:

`@Rule`

```
public ActivityTestRule<MainActivity> activityTestRule = new
ActivityTestRule<MainActivity>(MainActivity.class) {
```

```
    @Override
    public void beforeActivityLaunched() {
        super.beforeActivityLaunched();
        // code we'd like to execute to edit SharedPreferences
    }
};
```

How to get the app context:

```
Context context = InstrumentationRegistry.getInstrumentation().getTargetContext()
```

How to get access to shared preferences editor:

```
SharedPreferences.Editor editor =
```

```
context.getSharedPreferences(context.getPackageName(), Context.MODE_PRIVATE).edit();
```

### *Tip*

Editing the shared preferences is the same in a test as it is in the app.

## Bonus Exercise: Intended Intents

### *Intents, an introduction*

As explained earlier, apps consist of Activities, and Activities can invoke other activities (from the same app, but public activities from other apps as well). They do that with 'intents'. An intent is an abstract description of an operation to be performed. An intent has two parts: the type of action to be performed and data expressed as a Uri. An intent to open a website from our app (in another app, such as chrome) would look like this:

```
Intent(Intent.ACTION_VIEW, Uri.parse("https://saucelabs.com/resources"))
```

Android knows which apps have the ability to handle "ACTION\_VIEW" intents and will open your default app or ask you to select from a list of eligible apps.

### *Testing intents*

If you want to test if your app launches the right intent and contains the right data, you could check if a browser is opened and the right website is opened.

That means you have to automate the browser too. If we go that route, we're not just testing our app, but also testing Android's ability to process intents, and testing our browser's ability to process an incoming intent and open a website. To prevent this broadening of scope, Espresso offers the ability to validate and stub intents.

### *Validating intents*

Instead of an `ActivityTestRule` to launch an activity, we'll have to use an `IntentsTestRule`:

```
@Rule
public IntentsTestRule<MyActivity> intentsTestRule =
    new IntentsTestRule<>(MyActivity.class);
```

`IntentsTestRule` is an extension of the `ActivityTestRule`, this means that we can replace our `ActivityTestRule` with the `IntentsTestRule` and all our tests should keep working.

To check an intent, we can use the `intended()` method, example:

```
onView(withId(R.id.example_button_that_launches_intent)).perform(click());

intended(allOf(
    hasAction(equalTo(Intent.ACTION_VIEW)),
    hasData(hasHost(equalTo("https://saucelabs.com "))),
    hasData(hasPath(equalTo("/resources ")))));
```

The (other) available `IntentMatchers` are listed in the Espresso API cheat sheet.

source: <https://developer.android.com/reference/android/content/Intent.html>

### *Exercise*

In the Order Overview screen of the MailOrderCoffee app there's a submit button that sends out an intent to open an email app. We'd like to make sure that the correct intent gets launched.

### *Goal*

Learn how to keep the scope limited to the app by capturing and validating intents.

### *What*

Try to write a test where you validate the type and data of the intent.

### *How*

The intent is of type:

- `Intent.ACTION_SENDTO`

It contains extra's (data) of the following types:

- `Intent.EXTRA_EMAIL` (extra email addresses)
- `Intent.EXTRA_SUBJECT` (email subject)
- `Intent.EXTRA_TEXT` (email body text)

Use the `intent.hasExtra()` method to validate the intent.

### *Tip*

Keep it simple at first. Try to write the easiest test you can imagine. If you get that working, you can try testing for more properties of the intent. One by one.

## The app (features)

The MailOrderCoffeeShop app has 4 parts: onboarding, custom order, menu and order overview.

### *Onboarding*

The onboarding showcases the features of the app.

- There's an 'x' in the top left that dismisses the onboarding.
- You can swipe to view different onboarding screens.
- You can use the 'next arrow' in the bottom right to view the next onboarding screen.
- The 'next arrow' makes place for the 'done icon' on the last onboarding screen, which dismisses the onboarding.
- After being dismissed the onboarding will not be started again. Only after a clean install or after all app settings have been deleted.

### *Custom Order*

The custom order screen allows you to select different ingredients for your beverage.

- There are two buttons at the top, one for custom order and the other for the menu.
- The custom order button is disabled while on the custom order screen.
- The amount of espresso shots is controlled using the "+" and "-" buttons
- A counter shows the currently selected amount of shots
- If you try to select less than 0 shots (using the "-" button), a toast will be displayed.
- You can choose whether you'd like to have hot or cold espresso shots by using the "hot switch".
- The "hot switch" is set to true/on by default
- There's a spinner when you can select a type of milk to be added
- If the spinner is set to anything other than "no milk", you can select the milk preparation using radio buttons.
- If the milk type is set to custom % you get the ability to select a fat % using a seekbar (slider)
- The 'review order button' takes you to the 'order overview' screen
- Each order has to have at least 1 espresso shot: If you try to go to the order overview with 0 shots, a toast will be displayed.

### *Menu*

The menu screen allows you to pick a beverage from a scrollable list.

- There are two buttons at the top, one for custom order and the other for the menu.
- The menu button is disabled while on the menu screen.
- The list should be scrollable
- Each item on the list should have an icon, name and volume (in ml.)
- The icon should represent the beverage volume: 0-60ml == low, 61-180 == medium, 180+ == full.
- Clicking on an item should take you to the order overview screen.

### *Order Overview*

Allows you to review the order and submit the order using email



- Shows the beverage name for items from the menu
- The cup image should represent the beverage volume: 0-60ml == low, 61-180 == medium, 180+ == full
- The ingredients list should show all the ingredients, one for each line
- You can enter your name in the 'name edit text'
- You can enter your email address in the 'email edit text'
- If it's a custom order there's an additional field where you can give your order a name.
- The submit button launches an email app and supplies the details for you.
- Entering your name is mandatory, if it's a custom order the order name is also mandatory. If you press submit while these fields are empty, error messages are displayed below the text fields.