

# Espresso Patronum: The Magic of the Robot Pattern

---

Adam McNeilly: Android Engineer - OkCupid

# What is Espresso?

---

---

# Use Espresso to write concise, beautiful, and reliable Android UI tests<sup>1</sup>.

---

---

<sup>1</sup> <https://developer.android.com/training/testing/espresso/index.html>

# Three Classes To Know

1. `ViewMatchers`
2. `ViewActions`
3. `ViewAssertions`

# ViewMatchers

- `withId(...)`
- `withText(...)`
- `isFocusable()`
- `isChecked()`

# ViewActions

- `typeText(...)`
- `scrollTo()`
- `swipeLeft()`
- `click()`

# ViewAssertions

- `matches(Matcher)`
- `isLeftOf(Matcher)`
- `doesNotExist()`

# Espresso Cheatsheet<sup>2</sup>

**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)   
isDescendantOfA(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**)**  
.DataOptions  
.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(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)

<sup>2</sup> <https://developer.android.com/training/testing/espresso/cheat-sheet.html>

@AdamMc331  
@NYAndroidMeetup

8

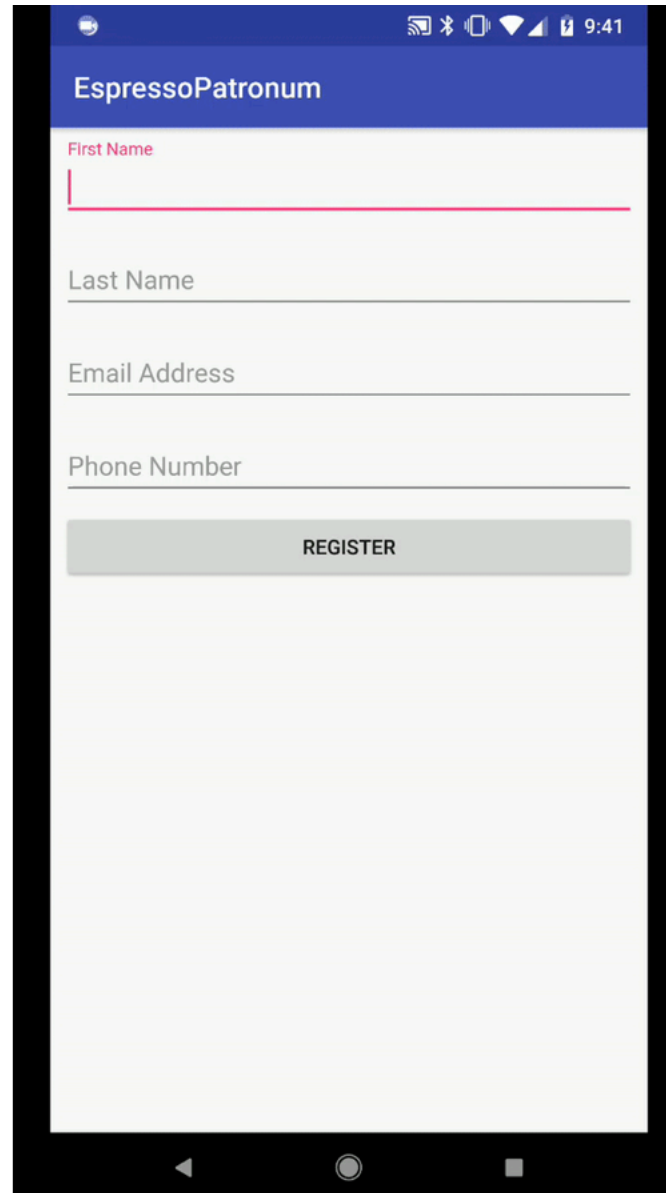


# Espresso Example

```
// onView gives us a ViewInteraction where we can perform an action
// or check an assertion.
onView(ViewMatcher)
    .perform(ViewAction)
    .check(ViewAssertion)

// Type into an EditText, verify it appears in a TextView
onView(withId(R.id.etInput)).perform(typeText("Adam"))
onView(withId(R.id.tvOutput)).check(matches(withText("Adam")))
```

# Sample Project



The screenshot displays a mobile application interface for 'EspressoPatronum'. The app has a blue header bar with the title 'EspressoPatronum'. Below the header, there is a registration form with four text input fields: 'First Name' (with a red label), 'Last Name', 'Email Address', and 'Phone Number'. Each field has a corresponding underline. Below the input fields is a grey button labeled 'REGISTER'. The app is running on an Android device, as indicated by the status bar at the top showing the time as 9:41 and various icons, and the navigation bar at the bottom.

# Test Successful Registration

```
@Test
fun testSuccessfulRegistration() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("amcneilly@okcupid.com"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.tvFullName)).check(matches(withText("Adam McNeilly")))
    onView(withId(R.id.tvEmailAddress)).check(matches(withText("amcneilly@okcupid.com")))
    onView(withId(R.id.tvPhoneNumber)).check(matches(withText("(123)-456-7890")))
}
```

# Test A Missing Field

```
@Test
fun testMissingEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    // onView(withId(R.id.etEmail)).perform(typeText("amcneilly@okcupid.com"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter an email address.")))
}
```

# One More Negative Test

```
@Test
fun testInvalidEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("blahblah"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter a valid email address.")))
}
```

# All Together

```
@Test
fun testSuccessfulRegistration() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("amcneilly@okcupid.com"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.tvFullName)).check(matches(withText("Adam McNeilly")))
    onView(withId(R.id.tvEmailAddress)).check(matches(withText("amcneilly@okcupid.com")))
    onView(withId(R.id.tvPhoneNumber)).check(matches(withText("(123)-456-7890")))
}

@Test
fun testMissingEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter an email address.")))
}

@Test
fun testInvalidEmailError() {
    onView(withId(R.id.etFirstName)).perform(typeText("Adam"))
    onView(withId(R.id.etLastName)).perform(typeText("McNeilly"))
    onView(withId(R.id.etEmail)).perform(typeText("blahblah"))
    onView(withId(R.id.etPhone)).perform(typeText("1234567890"))
    onView(withId(R.id.registerButton)).perform(click())

    onView(withId(R.id.etEmail)).check(matches(hasErrorText("Must enter a valid email address.")))
}
```

# Downfalls Of This Approach

1. Extremely Verbose
2. Unreadable
3. Not Easily Maintainable - What if a view changes?

# Introducing Robots

A robot is the middle man between your view and your code. This is a way of separating concerns just like an MVC/MVP/MVWTF architecture does with your application's code.



# Usage

```
@Test
fun testSuccessfulRegistration() {
    RegistrationRobot()
        .firstName("Adam")
        .lastName("McNeilly")
        .email("amcneilly@okcupid.com")
        .phone("1234567890")
        .register()
        .assertFullNameDisplay("Adam McNeilly")
        .assertEmailDisplay("amcneilly@okcupid.com")
        .assertPhoneDisplay("(123)-456-7890")
}
```

# Define ViewMatchers

```
class RegistrationRobot {  
    companion object {  
        private val FIRST_NAME_INPUT_MATCHER = withId(R.id.etFirstName)  
        private val LAST_NAME_INPUT_MATCHER = withId(R.id.etLastName)  
        private val EMAIL_INPUT_MATCHER = withId(R.id.etEmail)  
        private val PHONE_INPUT_MATCHER = withId(R.id.etPhone)  
        private val REGISTER_INPUT_MATCHER = withId(R.id.registerButton)  
  
        private val FULL_NAME_DISPLAY_MATCHER = withId(R.id.tvFullName)  
        private val EMAIL_DISPLAY_MATCHER = withId(R.id.tvEmailAddress)  
        private val PHONE_DISPLAY_MATCHER = withId(R.id.tvPhoneNumber)  
    }  
}
```

# Each Action As A Method

```
class RegistrationRobot {  
  
    fun firstName(firstName: String): RegistrationRobot {  
        onView(FIRST_NAME_MATCHER).perform(clearText(), typeText(firstName), closeSoftKeyboard())  
        return this  
    }  
  
    fun register(): RegistrationRobot {  
        onView(REGISTER_INPUT_MATCHER).perform(click())  
        return this  
    }  
  
    fun assertFullNameDisplay(fullName: String): RegistrationRobot {  
        onView(FULL_NAME_DISPLAY_MATCHER).check(matches(withText(fullName)))  
        return this  
    }  
  
    ...  
}
```

# Benefits

1. Readability
2. Maintainability
3. Better Test Reporting

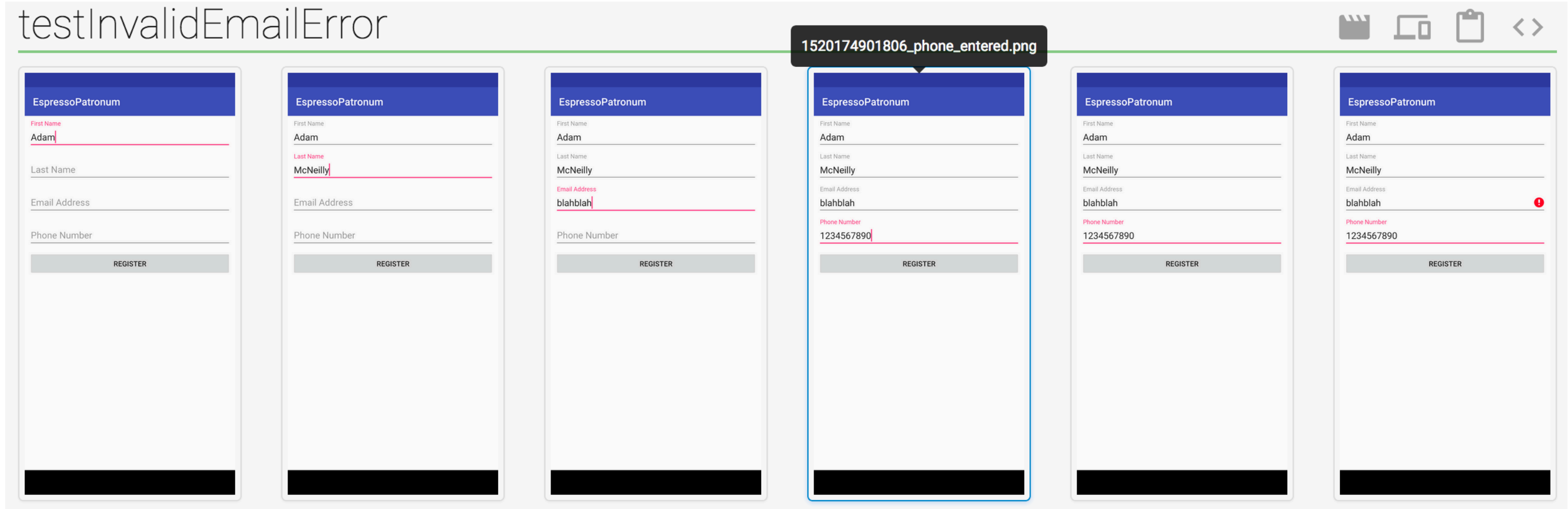
# Better Test Reporting Using Spoon<sup>3</sup>

Spoon will run all of our instrumentation tests and build us a static HTML report at the end.

---

<sup>3</sup> <https://github.com/square/spoon>

# Example Spoon Report



# When To Take Screenshots

- After assertions
- Before actions - unless that action leads to another screen
- On failure

# Adding Screenshots To Our Robot

```
fun firstName(firstName: String): RegistrationRobot {
    onView(FIRST_NAME_INPUT_MATCHER).perform(clearText(), typeText(firstName), closeSoftKeyboard())
    takeScreenshot(spoon, "first_name_entered")
    return this
}

fun register(): RegistrationRobot {
    takeScreenshot(spoon, "register_clicked")
    onView(REGISTER_INPUT_MATCHER).perform(click())
    return this
}

fun setFailureHandler(spoon: SpoonRule, context: Context) {
    Espresso.setFailureHandler { error, viewMatcher ->
        takeScreenshot(spoon, "test_failed")
        DefaultFailureHandler(context).handle(error, viewMatcher)
    }
}
```



# Takeaways

1. Use the robot pattern to make your tests more maintainable.
2. Your actual tests become easier and quicker to write once you've created a robot.
3. Robots can be leveraged for additional and more thorough.
4. This idea is not specific to Espresso or Spoon.

# Repository

— <https://github.com/AdamMc331/EspressoPatronum>