Andr0

Zeste de savoir

Introduction aux tests Android avec Espresso

12 février 2021

Table des matières

Le contenu de ce tutoriel est devenu obsolète et devrait être archivée et/ou supprimée.

La pratique d'effectuer des tests, peu importe sa forme, n'est pas la plus répandue aussi bien pour les développeurs amateurs qu'expérimentés. Les raisons peuvent être diverses et variées. Pour les amateurs, ils préfèrent souvent apprendre le développement "standard" dirons-nous, alors que les professionnels n'auront souvent "pas le temps" dans le délai accordé par leur chef de projet.

Alors pourquoi est-ce important de tester ses applications? Les avantages sont nombreux mais il est possible de les résumer en quelques mots: éviter les régressions et s'assurer du bon fonctionnement de son application à tout moment malgré les modifications apportées dans le temps. D'ailleurs, du côté professionnel, cette pratique commence doucement à intéresser les entreprises si elles parviennent à prouver et vendre l'intérêt des tests. Pour les amateurs, peut-être est-ce parce qu'il n'existe pas assez de tutoriels à ce sujet?

Dans ce cas, si vous êtes désireux de savoir comment tester votre application sur Android mais que vous disposez déjà de bonnes connaissances dans ce domaine, ce tutoriel est fait pour vous!

1. Présentation d'Espresso

Avant de présenter ne serait-ce qu'une technologie, il faut expliquer les grandes tendances dans les tests possibles et le type de test détaillé dans cette publication. Il existe 4 niveaux de test:

— Le test **unitaire** est une procédure qui vérifie une partie précise d'un logiciel et s'abstrait de tout le reste dans l'application. Par exemple, une application qui va calculer des taux immobiliers, des tests unitaires pourraient porter sur le calcul de ces taux. Cela oblige le développeur à bien modulariser son code. Il va différencier son interface de sa logique et pourra alors la tester.

- Le test d'**intégration** consolide plusieurs modifications pour vérifier s'il n'y a aucune erreur pendant la compilation de toutes ces modifications. Il exécute des vérifications et des analyses de code statique en bonus pour indiquer aux développeurs des erreurs ou avertissements potentiellement graves dans leurs applications. Ce test est souvent utilisé avec un outil de versionning comme Git et un serveur d'intégration comme Jenkins.
- Le test **fonctionnel** est un scénario utilisateur appliqué sur un écran de l'application. Il simule des interactions utilisateurs et fait des vérifications sur les données affichées sur les composants graphiques qui constitue l'interface après le test.
- Le test d'**acceptation** vise à assurer que le projet est conforme aux spécifications. Cette étape implique la présence d'une maitrise d'œuvre (entité apportant ses compétences techniques au besoin) et d'une maitrise d'ouvrage (entité porteuse du besoin) en effectuant des procédures de tests fonctionnels et techniques. Ce test arrive à la fin du projet.

Tous ces tests sont possibles dans le développement Android. La plupart sont même hautement recommandés en entreprise, notamment le test d'intégration et le test d'acceptation. Cependant, la publication portera sur le test fonctionnel.

Espresso est une bibliothèque développée par Google et présentée à l'occasion de l'édition 2013 de sa conférence Google I/O. Elle est encore toute jeune, sa communauté a besoin de grossir mais elle est déjà fonctionnelle, applicable à beaucoup d'applications et elle a pour vocation d'être intégrée directement dans le Kit de développement Android une fois arrivée à maturité. Par conséquent, il n'est pas inutile de commencer dès maintenant son apprentissage et de l'intégrer dans vos projets passés, présents et futurs.

Espresso se veut simple à utiliser et rapide à exécuter. Elle encourage les développeurs à se mettre à la place de l'utilisateur et de simuler leurs interactions potentielles en créant des scénarios d'utilisation. Ces scénarios regroupent un certain nombre d'actions, notamment la récupération d'un composant à l'écran, faire défiler une liste, cliquer sur un composant ou procéder à un enchaînement d'écrans.

D'autres bibliothèques existaient déjà avant, comme Robolectric ou Robotium . Ces deux bibliothèques sont connues auprès de la communauté Android et largement utilisées. Espresso a de l'intérêt parce qu'elle a l'avantage d'être petite, prédictible, facile à apprendre et à prendre en main. Elle teste les états attendus d'un composant, leurs interactions et effectue des vérifications sans se soucier d'attendre les enchaînements des écrans, leurs synchronisations, le temps de réponse parfois long de certaines instructions et laisse Espresso gérer toutes ces problématiques.

A l'occasion de la conférence Google, elle a été accueillie comme un renouveau dans les tests fonctionnels sous Android jusqu'à présent longs, ennuyeux et difficiles à développer.

2. Fonctionnement général

L'utilisation d'Espresso se veut la plus simple et le plus naturelle possible. Elle utilise des concepts utilisés dans d'autres bibliothèques de différents langages comme les appels à des méthodes statiques du type Hamcrest et leurs enchainements pour avoir un langage presque naturel. Cela a l'énorme avantage qu'un développeur A et un développeur B comprendront exactement de la même façon le code du test.

Espresso possède plusieurs composants généraux:

— `Espresso` est la classe qui sert de point d'entrée vers le test à venir. Elle propose les méthodes `onView` et `onData` qui, à elles seules, couvrent bons nombres de tests possibles sur une interface donnée.
— `ViewMatchers` contient une collection d'objets qui implémente l'interface `Matcher<? super View>`. Cela permet de récupérer une vue d'un écran et d'effectuer des actions et des vérifications dessus.
— `ViewActions` contient une collection d'objets `ViewAction` pour effectuer des actions sur une vue. Ces actions sont passées à la méthode `ViewInteraction.perform` et peuvent contenir plusieurs actions. Par exemple, il est possible de récupérer une vue, de défiler sur l'écran jusqu'à la voire et de cliquer dessus.
— `ViewAssertions` contient une collection de `ViewAssertion` pour effectuer des vérifications sur les vues. Ces vérifications sont passées à la méthode `ViewInteraction.check`. C'est à cette étape que le développeur testera l'état de sa vue après les actions qu'il aura fait dessus.

Les tests fonctionnels sur Android doivent s'exécuter sur une instance du système Android; c'est-à-dire qu'il est obligatoire de brancher un terminal physique en USB sur votre machine de développement ou de lancer un émulateur. Avant l'exécution des tests fonctionnels, l'environnement de travail propose de choisir un terminal physique ou virtuel pour exécuter les tests. Les différents tests s'exécuteront alors et s'afficheront visuellement à l'écran. C'est une des petites contraintes d'Android qui pose des problèmes notamment lorsqu'il est demandé de coupler les tests fonctionnels avec des tests d'intégrations. Mais ceci n'est pas l'objet de cette publication.

Sur l'environnement de travail Android Studio, les classes de test doivent se trouver sous l'arborescence suivante: `src > androidTest > com.example.package.tests`. `com.example.package` étant le package précisé dans l'attribut `package` de l'élément `manifest` dans le fichier AndroidManifest. En ce qui concerne Eclipse, il faut créer un second projet de test et le lier à l'application.

Il faut aussi savoir que les exemples dans les sections suivantes présenteront du code avec des imports statiques; c'est-à-dire que le code suivant:

```
1 onView(withId(R.id.main_b))
2         .perform(click())
3         .check(matches(withText(R.string.button_after)));
```

Ressemble en fait au code suivant:

```
1 Espresso.onView(ViewMatchers.withId(R.id.main_b))
2         .perform(ViewActions.click())
3
              .check(ViewAssertions.matches(ViewMatchers.withText(R.string.button
```

Il faut bien avouer que le premier code est bien plus lisible que le second. Comme les appels aux méthodes d'Espresso sont des appels à des méthodes statiques, Java permet de rajouter à sa liste d'import les méthodes statiques et ne plus devoir renseigner leurs provenances. Au dessus de la déclaration de votre classe, il figurera les imports suivants:

```
import static
    com.google.android.apps.common.testing.ui.espresso.Espresso.onView;
import static
    com.google.android.apps.common.testing.ui.espresso.action.ViewActions.*;
import static
    com.google.android.apps.common.testing.ui.espresso.assertion.ViewAssertions
import static
    com.google.android.apps.common.testing.ui.espresso.matcher.ViewMatchers.*;
```

3. Prérequis avant le développement des tests

Avant de commencer le développement des tests fonctionnels, il existe des prérequis qui doivent être respectés pour pouvoir les exécuter. Dans un premier temps, sachez qu'il existe un exécuteur ("runner" en anglais) qui est utilisé lors de l'exécution des tests unitaires ou fonctionnels. Espresso utilise un nouveau exécuteur nommé `GoogleInstrumentationTestRunner`. Pour pouvoir l'utiliser automatiquement pour tous les tests, un élément `instrumentation` doit être déclaré dans l'`AndroidManifest` de l'application Android comme fils de l'élément manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.randoomz.espressosample">

    <!-- The targetPackage must be the same package that the
        package on the manifest element -->
    <instrumentation

            android:name="com.google.android.apps.common.testing.testrunner.Goo
        android:targetPackage="org.randoomz.espressosample" />

    <application>
        <!-- Activities of the application ... -->
    </application>

</manifest>
```

Seconde chose, chaque classe de test doit étendre la classe abstraite `ActivityInstrumentationTestCase2` et fournir en type générique l'`Activity` qui sera utilisée par défaut pour les tests; c'est-à-dire qu'au début de chaque test, ce sera l'`Activity` renseignée dans ce type générique qui sera affichée à l'écran, prêt à être testée. Il faut aussi renseigner la classe au niveau du constructeur et la renvoyer à la classe mère via l'instruction `super()`.

Par contre, cette `Activity` n'est pas appelée par le framework de test, c'est au développeur de l'afficher pour la rendre testable. Si elle doit s'afficher à chaque démarrage de chaque test, il

suffit d'appeler la méthode synchrone `T getActivity()` (T prenant la valeur du type générique spécifié au niveau de la classe abstraite).

```
1  public class MainActivityTest extends
       ActivityInstrumentationTestCase2<MainActivity> {
2      public MainActivityTest() {
3          super(MainActivity.class);
4      }
5
6      @Override
7      public void setUp() throws Exception {
8          super.setUp();
9          // Call the activity before each test.
10         getActivity();
11     }
12
13     // Our tests...
14 }
```

Quant aux méthodes de test, contrairement aux tests unitaires, il n'y a aucune annotation à spécifier. Il suffit de les déclarer les unes à la suite des autres pour que l'exécution des tests s'effectuent dans l'ordre où elles sont déclarées.

```
1  public void testNextActivity() {
2      // Our test...
3  }
```

i Par convention, une classe de test termine par le mot "Test" et une méthode de test commence par le mot "test". C'est une convention qui existe avec les tests unitaires et repris pour les tests fonctionnels avec Espresso.

4. Des tests simples avec onView

4.1. Développement piloté par les tests

La pratique de développer des tests pour ses applications n'est pas répandue. Tout le monde se dit qu'il faut en faire mais les développeurs qui le font vraiment ne font pas partie d'une majorité. Alors qu'en est-il des techniques de développement dirigées par les tests? Clairement, c'est encore moins utilisé que le développement des tests eux-même mais c'est une pratique intéressante à connaitre.

Le TDD (Test Driven Development ou développement piloté par les tests en français) est l'une de ces techniques de développement dirigées par les tests les plus connues. L'idée est simple,

développer les tests avant même de commencer le développement du logiciel. Une fois les tests développés, l'objectif est de faire passer tous les tests au vert[1]. Les avantages sont multiples: Pour un développeur A qui commence à faire passer les tests au vert et qui s'arrête en cours de route, un développeur B saura exactement où en était le développeur A (en exécutant la suite de tests) et pourra continuer sa tâche; Le code pour faire passer le test au vert est, généralement, le plus simple et concis possible.

L'approche TDD possède un cycle de développement extrêmement simple:

1. Écrire un premier test;
2. Vérifier qu'il échoue (car le code qu'il teste n'existe pas), afin de vérifier que le test est valide;
3. Écrire juste le code suffisant pour passer le test;
4. Vérifier que le test passe;
5. Refactoriser le code; c'est-à-dire l'améliorer tout en gardant les mêmes fonctionnalités.

Cette méthode a été utilisée dans le développement des exemples de cette publication et sera utilisée dans les explications qui seront données.

4.2. Tester le changement d'un bouton

Premier test, il faut pouvoir vérifier qu'un bouton modifie son texte lorsque l'utilisateur clique dessus. Pour se faire, il faut pouvoir: récupérer la vue du bouton; vérifier son texte de départ; effectuer l'action du clique dessus; et, vérifier si le texte a bien été changé.

La première étape consiste à récupérer la vue du bouton.

```
1 onView(withId(R.id.main_b))
```

Via la méthode de la classe `Espresso`, la vue est récupérée avec la méthode `onView` et elle est récupérée grâce à un identifiant avec la méthode `withId` de la classe `ViewMatchers`. La seconde étape consiste à vérifier que le texte initial est correct.

```
1 onView(withId(R.id.main_b)).check(matches(withText(R.string.button_before)));
```

Après la récupération de la vue du bouton, la méthode `check` effectue une vérification. Pour effectuer cette vérification, la méthode la plus couramment utilisée est `matches`. Sur la vue récupérée, elle va appliquer le `Matcher` qu'elle a en paramètre. Dans le cas présent, elle va vérifier si le bouton possède le texte renseigné dans la méthode `withText`.

Une fois cette vérification terminée, il faut cliquer sur le bouton et vérifier que son texte a bien changé. Pour se faire, il faut récupérer la vue du bouton comme pour la vérification précédente et cliquer dessus.

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

La méthode `perform`, de la classe `ViewInteraction`, est utilisée pour effectuer des actions sur des vues. Pour le test, il faut cliquer dessus. La méthode `click`, de la classe `ViewActions`, est renseignée en paramètre.

La méthode `perform` est la seule méthode qui peut renseigner plusieurs paramètres pour effectuer plusieurs actions. Par exemple, si le bouton n'est pas affiché à l'écran et qu'il faut d'abord défiler sur l'écran pour le voir, il faudra défiler puis cliquer.

```
onView(...).perform(scrollTo(), click());
```

Pour finir, il faut effectuer une vérification à la suite de l'action qui se fera toujours grâce à la méthode `check` et se positionne à la suite de la méthode `perform`. Sauf que cette fois, le texte du bouton doit avoir la valeur **après** le clic.

```
onView(withId(R.id.main_b)).perform(click()).check(matches(withText(R.string.bu
```

Le test est maintenant écrit, si le cycle du TDD est respecté, il faut exécuter le test et vérifier qu'il passe au rouge. Pour exécuter un test, il suffit de faire un clic droit sur le fichier de test et l'exécuter en cliquant sur "Run". Sans surprise, le test est au rouge puisqu'il ne peut même pas compiler le code source. Il ne parvient pas à trouver d'identifiant `R.id.main_b` ni les chaînes de caractères `R.string.button_before` et `R.string.button_after`.

Les chaînes renseignées dans le fichier `string.xml` du projet n'ont pas d'importance. Il suffit de rajouter les ressources manquantes et de remplir des valeurs différentes pour les deux. En ce qui concerne le bouton, il faut créer un nouveau layout avec un bouton dedans qui renseigne l'identifiant voulu par les tests.

Une fois le layout et l'`Activity` associée créés, il faut attacher un listener `OnClickListener` au bouton et changer son texte avec la nouvelle valeur. La classe associée ressemblera à l'exemple ci-dessous.

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final Button button = (Button) findViewById(R.id.main_b);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
```

```
10              public void onClick(View v) {
11                  button.setText(R.string.button_after);
12              }
13          });
14      }
15  }
```

Si le test est exécuté une seconde fois, il passera au vert.

4.3. Tester le changement d'écran

Espresso tente d'abstraire tous les temps d'attente derrière les actions utilisateurs. Plus besoin de s'ennuyer avec des synchronisations, des `sleep` ou d'autres instructions du même style souvent utilisées avec les bibliothèques de test existantes. Lorsque l'utilisateur passe d'un écran à un autre, Espresso se charge tout seul d'attendre le temps nécessaire avant d'effectuer les vérifications voulues par le développeur.

Ce second test doit vérifier qu'une action sur un bouton lance une nouvelle `Activity` et, en bonus, qu'une action sur le bouton retour du téléphone ramène à l'`Activity` initiale. Pour se faire, il faut pouvoir: récupérer le bouton; cliquer dessus; vérifier qu'un nouvel écran est lancé; revenir à l'écran précédent; et, vérifier que l'utilisateur se trouve sur l'écran initial.

La première étape consiste à récupérer le bouton et à cliquer dessus.

```
1  onView(withText(R.string.button_next_activity)).perform(click());
```

L'utilisateur sera censé se trouver sur une nouvelle `Activity`. Pour vérifier qu'il s'y trouve bien, il suffit de vérifier l'une des vues qui compose l'écran. La seconde étape consiste à récupérer et à vérifier le texte d'un `TextView` placé sur l'écran de la seconde `Activity`.

```
1  onView(withId(R.id.second_tv_welcome)).check(matches(withText(R.string.second_
```

La troisième étape consiste à simuler l'utilisation du bouton retour des terminaux pour revenir à l'écran précédent, qui est le fonctionnement par défaut du bouton. Pour se faire, la classe `Espresso` fournit une méthode supplémentaire.

```
1  pressBack();
```

La quatrième et dernière étape consiste à vérifier que l'utilisateur est bien revenu sur l'écran initial. Un `TextView` a été rajouté sur le premier écran et l'objectif est de vérifier qu'il est bien affiché.

```
onView(withId(R.id.main_tv)).check(matches(isDisplayed()));
```

Ce test ne compile pas encore à cause du TDD. Pour le faire fonctionner, il faut écrire les chaines de caractères appropriées renseignées par les identifiants utilisés et faire évoluer la précédente `Activity`. Les besoins du test nécessite l'ajout d'un bouton qui va ouvrir une seconde `Activity`. Rajoutez simplement un bouton dans `MainActivity` et attachez lui un `OnClickListener` qui lance `SecondActivity` pour le second écran.

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final Button button = (Button) findViewById(R.id.main_b);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                button.setText(R.string.button_after);
            }
        });

            findViewById(R.id.main_b_next_activity).setOnClickListener(new
            View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startActivity(new Intent(MainActivity.this,
                    SecondActivity.class));
            }
        });
    }
}
```

Quant à la seconde `Activity`, nul besoin de donner un exemple. Il suffit de la créer et d'attacher un layout avec un `TextView` et le contenu qui va bien pour faire passer le test fonctionnel au vert!

5. Des tests simples avec onData

L'instruction `onData` est un peu plus complexe que sa sœur, `onView`. Elle s'utilise avec des listes de tous les types. Par exemple, elle peut retrouver un élément dans une liste d'une `ListView` ou d'un `Spinner` puis, comme pour `onView`, effectuer des actions et des vérifications. A l'exception

1. Un test passe au vert lorsqu'il s'exécute correctement. Sinon, il est au rouge.

près qu'`onData` ne prend pas en paramètre un `ViewMatcher` adapté aux vues mais un `Matcher`, standard de la bibliothèque Hamcrest intégré dans Espresso.

Le test est le suivant: Dans une liste de pays, il faut cliquer sur le pays "Belgique" qui doit lancer une nouvelle `Activity` en marquant le nom du pays et pouvoir revenir en arrière avec la touche retour du téléphone. Pour se faire, il faut pouvoir: trouver le pays "Belgique"; cliquer dessus; vérifier qu'un nouvel écran est lancé; vérifier que le texte affiche bien "Belgique"; revenir à l'écran précédent; et, vérifier que l'utilisateur se trouve sur l'écran initial.

En fin de compte, la première étape est la seule inconnue dans ce test. Elle consiste à récupérer l'élément "Belgique" dans la liste de chaîne de caractères et de cliquer dessus.

```
1 final String item = "Belgique";
2 onData(allOf(is(instanceOf(String.class)),
      is(item))).perform(click());
```

Donc, pour tous les éléments (`allOf`) qui sont (`is`) du type `String` (`instanceOf`) et qui sont égaux (`is`) à la valeur "Belgique", clique dessus. C'est à la fois simple et puissant puisqu'il est inutile de se préoccuper de savoir si la liste a déjà chargée tous les éléments de la liste. Espresso va la parcourir et initialiser la première occurrence qui correspond à l'instruction pour effectuer l'action.

Les étapes suivantes sont connues de par le précédent exemple. Mais pour rappel, voici comment vérifier que le second écran affiche correctement une valeur dans un texte, revenir sur l'écran précédent et vérifier que l'utilisateur se retrouve bien à l'écran initial.

```
1 onView(withId(R.id.second_tv_welcome)).check(matches(withText(item)));
2 pressBack();
3 onView(withId(android.R.id.list)).check(matches(isDisplayed()));
```

Créez une nouvelle `Activity` pour accueillir la liste qui va lister les pays voulus et donc, sans oublier d'écrire ce nouveau test dans une nouvelle classe de test avec comme classe générique le nom donné à la nouvelle `Activity`.

```
1 public class ListViewActivity extends ListActivity {
2     public static final String ARG_FROM =
          "ListViewActivity.Key.From";
3
4     private final String[] countries = new String[]{
5             "Allemagne", "Argentine", "Belgique", "France",
                  "Italie", "Espagne"
6     };
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) {
```

```
10          super.onCreate(savedInstanceState);
11          setListAdapter(new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1, countries));
12      }
13
14      @Override
15      protected void onListItemClick(ListView l, View v, int
            position, long id) {
16          final Intent intent = new Intent(this,
                SecondActivity.class);
17          intent.putExtra(ARG_FROM, (String)
                getListAdapter().getItem(position));
18          startActivity(intent);
19      }
20  }
```

Faites évoluer `SecondActivity` pour faire passer le test au vert tout en gardant au vert le précédent test. Il est extrêmement rare de n'avoir qu'une classe de test dans toute son application. C'est pourquoi il faut régulièrement exécuter toutes les classes de test pour vérifier qu'il n'y a aucun test qui passe au rouge et éviter les régressions par la même occasion.

Ainsi s'achève cette introduction aux tests fonctionnels. En espérant qu'elle aura permis de convaincre les futurs développeurs que vous êtes. Et que vous aurez compris tous les avantages des tests fonctionnels et du développement orienté TDD.

Source: Site officiel d'Espresso

Liste des abréviations

TDD Test Driven Development. 5–7, 9, 11