Automated mobile UI testing with Xamarin.UITest and Xamarin Test Cloud

16 December 2014 | **Software Engineering** | **Kerry Lothrop**
Reading time: 5 minutes

Systematic automated testing of mobile apps is unfortunately still not a part of every project. I've seen many projects that only rely on manual exploratory testing. This is, in part, because the tools for automated testing of mobile apps haven't always been there.

At their first Evolve conference in 2013, Xamarin announced that they had acquired *LessPainful*, a company mostly known for the creation of the popular Calabash mobile testing toolkit. The other big testing-related announcement at Evolve was that of the closed beta of Xamarin Test Cloud, a very impressive cloud-based platform for testing apps on a wide range of Android and iOS devices.

Calabash

Calabash tests are typically written using a combination of Ruby and Gherkin. Because of my .NET background, I wanted to try the new Xamarin.UITest framework to write tests with C# and NUnit. To make things more complicated, I decided to test an Ionic/Cordova hybrid app instead of a fully native or Xamarin C# app.

Calabash works for iOS and Android. Operating the actual UI controls is accomplished through a Calabash component (Instrumentation Test Server) on the device or emulator that is controlled remotely by a test runner on the development machine or continuous integration server. For Android, the Instrumentation Test Server is deployed as a separate app. Due to more strict sandboxing on iOS, the Instrumentation Test Server needs to be integrated into the app in the form of the calabash.framework. While it's unfortunate that it's not possible to test the binary that's actually being deployed to the iOS App Store (calabash.framework uses private APIs that will not pass the automated App Store acceptance tests) the influence of the library on the app will typically not show in the tests.

Xamarin.UITest

To get started with Xamarin.UITest you'll have to set up a .NET test project. This can be done in Visual Studio for Android tests and in Xamarin Studio for iOS and Android (Xamarin is working on supporting testing iOS apps from Visual Studio). When the test starts it needs to know the path to the IPA or APK file to be used by all tests.

```
[SetUp]
public void SetUp()
{
    _app = ConfigureApp.iOS.AppBundle(PathToIPA).StartApp();
}
```

or

```
[SetUp]
public void SetUp()
{
    _app = ConfigureApp.Android.ApkFile(PathToAPK).StartApp();
}
```

The first unit test is typically started off like this:

```
[Test]
public void FirstUnitTest()
{
    _app.Repl();
}
```

This starts up a typically very useful C# REPL that can be used to identify the UI elements. When outputting all elements on iOS you will see something like this for a hybrid app:

```
Full log file:
/var/folders/wg/m0xyxrr10s5b4nz83dvgjssh0000gn/T/uitest/log-2014-12-13
_22-37-58-452.txt
iOS test running Xamarin.UITest version: 0.6.7
Initializing iOS app on http://127.0.0.1:37265/.
Test server version: 0.12.0.pre1
Running activated version with api key from ConfigureApp

App has been initialized to the 'app' variable.
```

```
Exit REPL with ctrl-c or see help for more commands.

>>> app.Query(c => c.All())
[
  {
    "Id":null,
    "Description":"<UIWindow: 0x7a7d2060; frame = (0 0; 320 568);
gestureRecognizers = <NSArray: 0x7a7d2950>; layer = <UIWindowLayer:
0x7a7d2620>>",
    "Rect":{
      "Width":320.0,
      "Height":568.0,
      "X":0.0,
      "Y":0.0,
      "CenterX":160.0,
      "CenterY":284.0
    },
    "Label":null,
    "Text":null,
    "Class":"UIWindow",
    "Enabled":true
  },
  {
    "Id":null,
    "Description":"<UIView: 0x7b8be890; frame = (0 0; 320 568);
autoresize = W+H; layer = <CALayer: 0x7b8bdd70>>",
    "Rect":{
      "Width":320.0,
      "Height":568.0,
      "X":0.0,
      "Y":0.0,
      "CenterX":160.0,
      "CenterY":284.0
    },
    "Label":null,
    "Text":null,
    "Class":"UIView",
    "Enabled":true
```

```
  },
  {
    "Id":null,
    "Description":"<UIWebView: 0x7baa5660; frame = (0 0; 320 568);
autoresize = W+H; layer = <CALayer: 0x7baa5770>>",
    "Rect":{
      "Width":320.0,
      "Height":568.0,
      "X":0.0,
      "Y":0.0,
      "CenterX":160.0,
      "CenterY":284.0
    },
    "Label":null,
    "Text":null,
    "Class":"UIWebView",
    "Enabled":true
  },
  {
    "Id":null,
    "Description":"<_UIWebViewScrollView: 0x7baaaed0; frame = (0 0;
320 568); clipsToBounds = YES; autoresize = H; gestureRecognizers =
<NSArray: 0x7baab610>; layer = <CALayer: 0x7baab200>; contentOffset:
{0, 0}; contentSize: {320, 568}>",
    "Rect":{
      "Width":320.0,
      "Height":568.0,
      "X":0.0,
      "Y":0.0,
      "CenterX":160.0,
      "CenterY":284.0
    },
    "Label":null,
    "Text":null,
    "Class":"_UIWebViewScrollView",
    "Enabled":true
  },
  {
```

```
    "Id":null,
    "Description":"<UIWebBrowserView: 0x7b2a3000; frame = (0 0; 320
568); gestureRecognizers = <NSArray: 0x7baa75e0>; layer = <UIWebLayer:
0x7b8c0e40>>",
    "Rect":{
      "Width":320.0,
      "Height":568.0,
      "X":0.0,
      "Y":0.0,
      "CenterX":160.0,
      "CenterY":284.0
    },
    "Label":null,
    "Text":null,
    "Class":"UIWebBrowserView",
    "Enabled":true
  },
  {
    "Id":null,
    "Description":"<UIImageView: 0x7a7d7700; frame = (0 565.5; 320
2.5); alpha = 0; opaque = NO; autoresize = TM; userInteractionEnabled
= NO; layer = <CALayer: 0x7a7d7af0>> - (null)",
    "Rect":{
      "Width":320.0,
      "Height":2.5,
      "X":0.0,
      "Y":565.5,
      "CenterX":160.0,
      "CenterY":566.75
    },
    "Label":null,
    "Text":null,
    "Class":"UIImageView",
    "Enabled":false
  },
  {
    "Id":null,
    "Description":"<UIImageView: 0x7a7d82d0; frame = (317.5 0; 2.5
```

```
568); alpha = 0; opaque = NO; autoresize = LM; userInteractionEnabled
= NO; layer = <CALayer: 0x7a7d8350>> - (null)",
    "Rect":{
      "Width":2.5,
      "Height":568.0,
      "X":317.5,
      "Y":0.0,
      "CenterX":318.75,
      "CenterY":284.0
    },
    "Label":null,
    "Text":null,
    "Class":"UIImageView",
    "Enabled":false
  }
]
```

Working with hybrid apps

The only thing to see here is the full-screen UIWebView. Since all UI elements are rendered inside this web view there is no relevant information that can be gathered through this. To see the actual view elements we'll need to look inside the web view. This is possible through Safari for iOS and through Chrome for Android. You can use the browser developer tools to visually identify the elements inside the DOM.

Once you've found the UI element you want to address you'll need to formulate a CSS selector to uniquely identify the element you want to call that works on both Chrome and Safari. A call to tap an element looks like this:

```
_app.Tap(c => c.WebView().Css("#element"));
```

Entering a text looks like this:

```
_app.EnterText(c => c.WebView().Css("#username"), "testuser");
```

It's important to make sure that an element you want to interact with is already available if it's on a subsequent page. This can be achieved by waiting for the element:

```
_app.WaitForElement(c => c.WebView().Css("#username"), "Timed out
waiting for login view");
```

You can also execute arbitrary JavaScript on the web view:

```
var backgroundColor = _app.Query(c =>
c.WebView().InvokeJS("document.getElementById('element-1').currentStyl
e.backgroundColor")).Single();
Assert.AreEqual("#0c9462", backgroundColor);
```

Xamarin.UITest offers swiping methods to trigger a swipe gesture but the performed swipe may not be enough to actually navigate between two full-screen pages. A manual swipe implementation may look like this:

```
private void SwipeLeft()
{
    var bodyRect = _app.Query(c => c.WebView()).Single().Rect;
    float fromCenterX = bodyRect.CenterX * 0.9f;
    _app.DragCoordinates(
        bodyRect.CenterX + fromCenterX,
        bodyRect.CenterY,
        bodyRect.CenterX - fromCenterX,
        bodyRect.CenterY);
}
```

Enter Test Cloud

The fun starts when you upload your app and corresponding test script to Xamarin Test Cloud and give it a spin on a multitude of devices. To prepare for this it makes sense to insert a number of named screenshots into each test case.

```
_app.Screenshot("After swipe");
```

In the Test Cloud portal you can define a set of devices to deploy to and run the test on all of them. I did this for a simple test of the Ahoi! train timetable app by my coworker Christian

Kohler. Ahoi! is based on Ionic and has a wonderful user experience that allows you to search European train timetables with only two taps.

The results are very impressive and allow deep insights into the behavior of an app on a large number of mobile devices.

Summary

Calabash and Xamarin.UITest are very powerful tools for UI testing of any kind of mobile app. Both the Gherkin/Ruby and C#/NUnit approaches allow testing almost any type of user interaction. If you combine these possibilities with Xamarin Test Cloud you possess the tools to provide your users with the maximum level of quality across a multitude of Android and iOS devices, OS versions and form factors. The downside is the lack of support for other platforms (e.g., Windows RT, Windows Phone) and the current price tag of Xamarin Test Cloud.