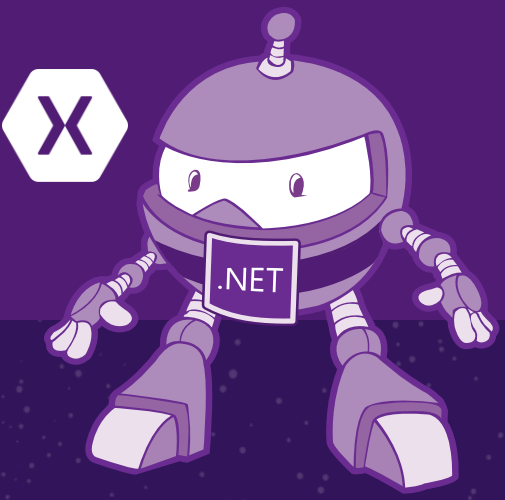

.NET Conf
"Focus on Xamarin"
.NET
focus.dotnetconf.net

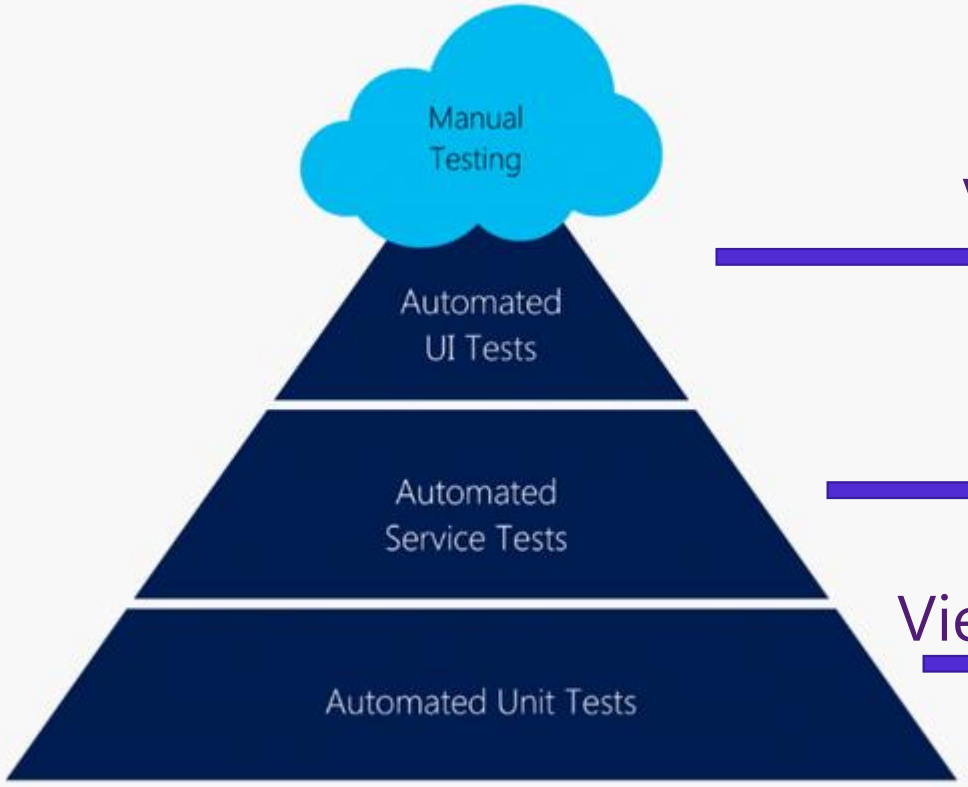
Testing your Xamarin Apps

Codrina Merigo
Software Engineer
Fresenius Medical Care



Intro

App Testing



View

ViewModel

UI Tests This is the layer where you would be testing the product more from an end-user's perspective. Your top priority would be to ensure that 'UI Design Flow' is in-line with the design requirements.

Service Tests helps you ensure that the services are working properly and give you the right data.

Unit Tests are normally written by the *developers* who write test code to verify the functionalities that were developed by them.

FIRST Principles

Fast - tests are slow but devs must try not to write them even slower

Independent - not use other tests

Repeatable - not use constants like access tokens

Self-validating - by using asserts

Timely - try not to be very complicated, maybe an action can be split in multiple tests

AAA Pattern

Arrange, Act, Assert

Sample

```
public int MySum(int a, int b)
{
     return a + b;
}
```

```
        [Test]
        public void TestMySum()
        {
            // Arrange
            int a = 5;
            int b = 7;

            // Act
            int result = MySum(a, b);

            // Assert
            Assert.AreEqual(12, result);
        }
```

Unit Testing

.NET

A unit test takes a **small unit of the app**, typically a method, isolates it from the remainder of the code, and verifies that it behaves as expected. Its goal is to check that each unit of **functionality** performs **as expected**, so that errors don't propagate throughout the app.

Why do we Unit Test?

Test system functionalities

Test small part of the application at once

Identify and simplify architecture

Identity bugs earlier

Avoid regressions

Unit Test Frameworks

Testing view models from MVVM applications is identical to testing any other classes and many .NET and c# Frameworks can be used.

nUnit

xUnit.net

VSTest

Unit test your own code

Use a mock (or moq) service to create instances of your interfaces.

When unit testing a Xamarin.Forms view model, don't focus on things like Bindings, JSON parsers, Plugins or purely Xamarin.Forms features as they are tested by the Xamarin.Teams.

Test your commands and logic from view models.

Based on the selected framework, use the proper test decorations.

IntelliTest can help you!

Interface everything

In your Unit Test project, you can add any plugin or nuget package you might use in your app and even Xamarin.Essentials.

Creating your own IXamarinEssentials, and exposing only the methods and properties you would like to use, gives more flexibility.

Sample

```
public class MockConnectivity : IConnectivity
        {
            public bool IsConnected => true;

            public IEnumerable<ConnectionType> ConnectionTypes => new[] { ConnectionType.WiFi };

            public IEnumerable<ulong> Bandwidths => new[] { (ulong)4000 };

            public event ConnectivityChangedEventHandler ConnectivityChanged;
            public event ConnectivityTypeChangedEventHandler ConnectivityTypeChanged;

            …

    }
```

Demo

UI Testing

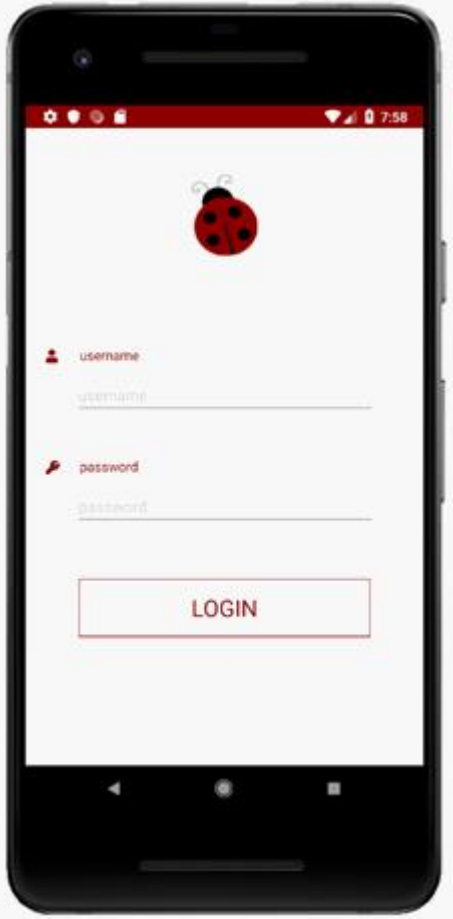
.NET

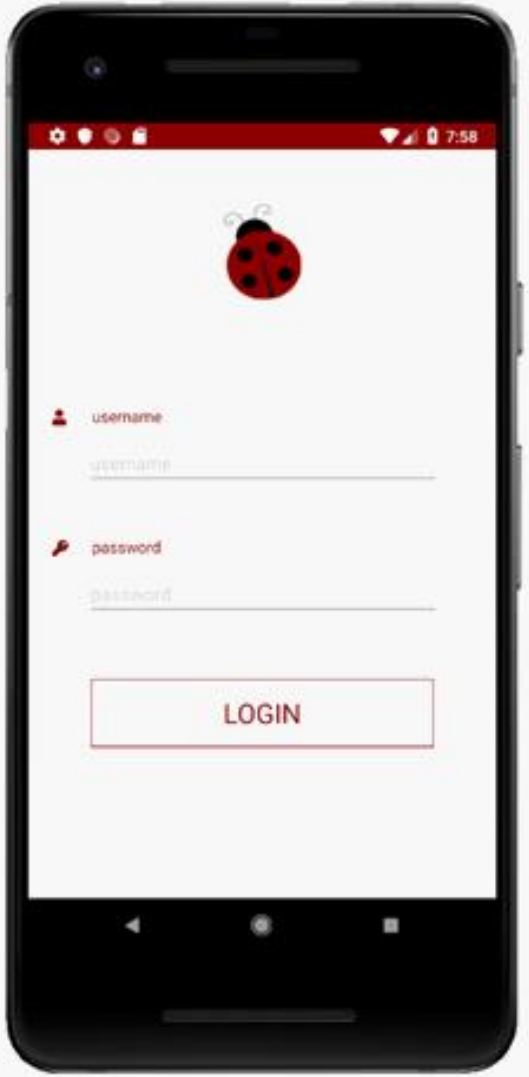
Since the testing at **UI level** is so **fragile**,
it is recommended to **focus** on these tests
to only verify '**UI flow & interactions**'
without looking into the system functionality.

How would you test UI manually?

E.g.: login.

1. *Enter username and password*
2. *Tap Login*
3. *Locate the username field -> interaction: type your name into the field*
4. *Repeat for the password*
5. *Locate and tap the Sign In button*
6. *Ensure you have logged in*





The two main tasks that you do manually are:

1. Locate the element you want and
2. Interact with it

You will do the same in your automated UI Test!

What should you UI Test for?

Verify your views and controls on the screen

Check for navigation between pages

Interact with ‘live’ elements like buttons, checkboxes, switches or tabs

BDD Approach

Behavior Driven Development

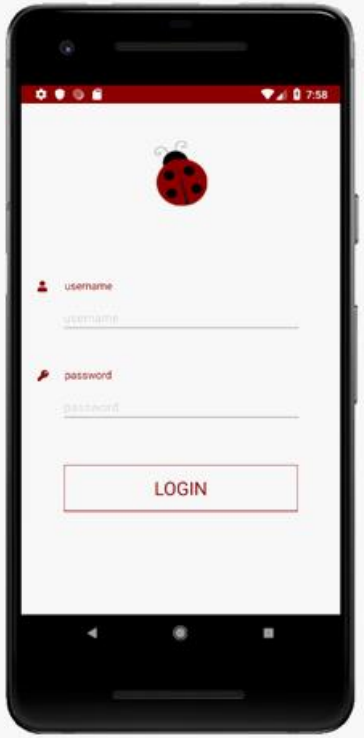
Given → When → Then

This process allows you to write tests in a *non-technical language* that everyone can understand (e.g. a domain-specific language like *Gherkin*). BDD forms an approach for building a **shared understanding** on what kind of software to build **by discussing examples.**

Sample

As a [role]
I want [feature]
So that [benefit]

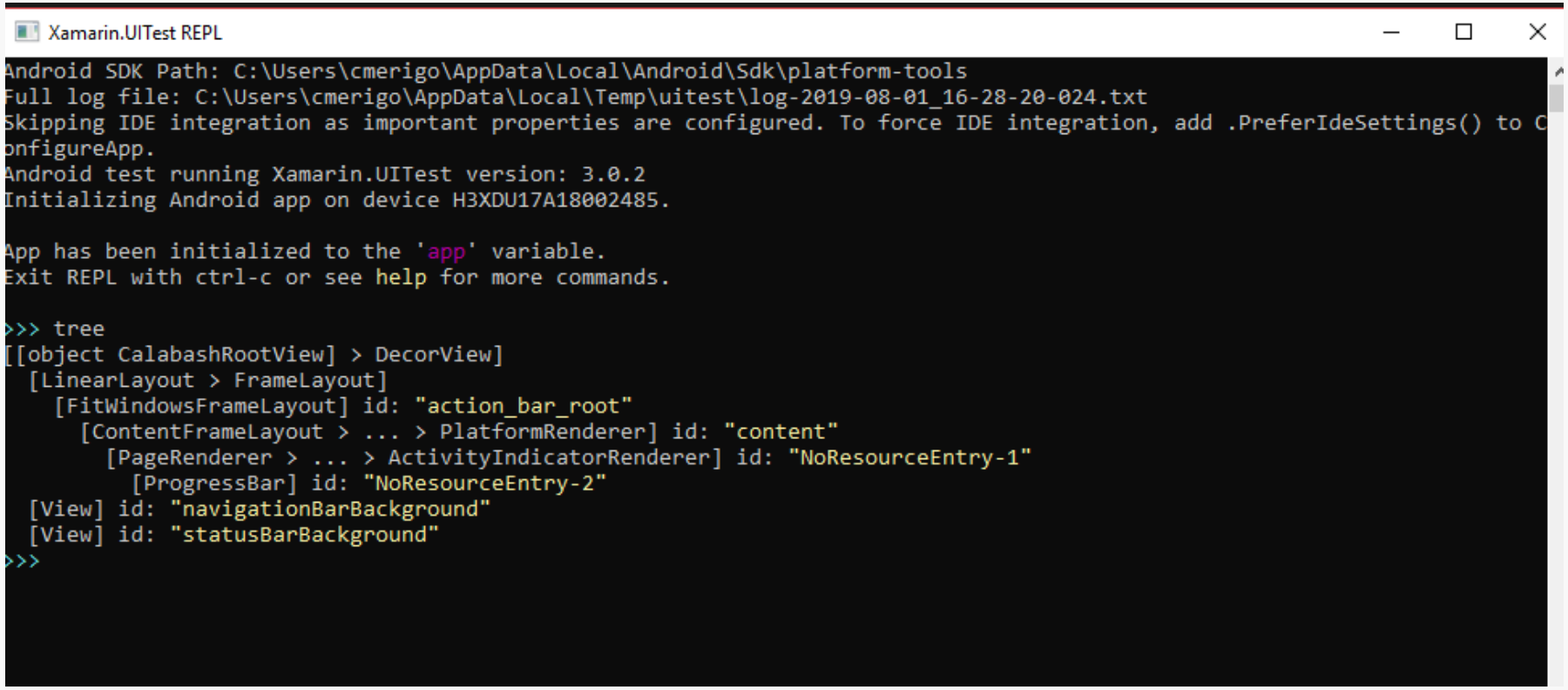
```
As a registered user,
I want to be able to login,
so that I can see the Application.

Scenario 1: User is able to log in.

Given that I am a registered user,
when I enter the username 'Codrina'
and password 'ladybug',
then I should see
     the first screen of the app.
```



UI Automation is supposed to run **slow** in order to emulate an actual user interaction - keep that in mind that sometimes we may wait for an element **to pop on** the screen before we verify if it **is actually on** the screen.

Repl() - read-eval-print-loop

The **REPL** is a console-like environment in which the developer enters expressions or commands.
It will then evaluate those expressions and display the results to the user.



tree
app.Query(e => e.All())

app.Flash(e=>e.All())

The REPL is helpful when creating UITests as it allows us to **explore the user interface** and create the queries and statements so that the test may interact with the application.

Demo

Meet the AutomationId

In order to *identify* a control on the screen, each control needs an unique identifier called **AutomationId** set to the visual element, for example:

```
<Label x:Name="label1" AutomationId="MyLabel" Text="Hello, Xamarin.Forms!" />
```

Or

```
var label1 = new Label {
    Text = "Hello, Xamarin.Forms!",
    AutomationId = "MyLabel"
};
```

It’s important also to write every single action and **not take anything for granted**

Queries inside and outside Repl()

Samples

```
//label
bool myLabel = app.Query(e => e.Marked(“MyLabel")).Any();
app.Tap(“MyLabel");


//swipe gestures
app.SwipeLeftToRight();

//entry
 app.Tap(“MyEntry");
 app.EnterText(Constants.Value);
 app.DismissKeyboard();

//inside webview
/*through Safari for iOS and through Chrome for Android use the browser developer
tools to visually identify the elements inside the DOM.*/
 app.Tap(c => c.WebView().Css("#element"));
```

Demo

Interacting with elements

Waiting

IApp.WaitForElement
IApp.WaitForNoElement

```
app.WaitForElement(c=>c.Marked(”MyLabel”)
,“Did not see MyLabel”.",
new TimeSpan(0,0,0,90,0));
```

Scrolling

IApp.ScrollDownTo

```
app.ScrollDownTo(c =>e.Marked(“MyLabel"));
```

Impersonate user

Gestures

IApp.DoubleTap – Two quick taps on the first matched view.
IApp.DragCoordinates – Continuous drag between two points.
IApp.PinchToZoomIn – Pinch gesture on the matched view to zoom in.
IApp.PinchToZoomOut – Pinch gesture on the matched view to zoom out.
IApp.ScrollUp / **IApp.ScrollDown** – Touch gesture that scrolls down or up.
IApp.SwipeLeftToRight / **IApp.RightToLeft** – Left-to-right or right-to-left gesture swipe.
IApp.Tap – Taps the first matched element.
IApp.TouchAndHold – Continuously touches the view.

```
app.DoubleTap(c=>c.Marked (“MyLabel"));
```

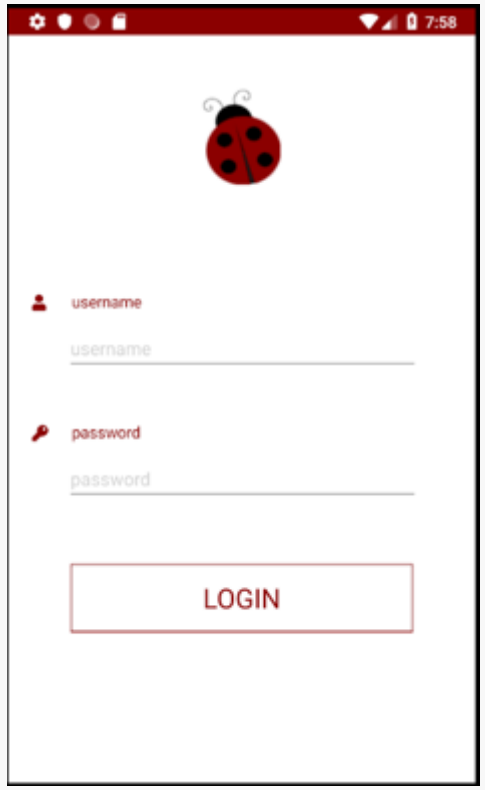
Screenshots

At any time, in your ui test, you can take a screenshot of the current view for further checks or you might want to take a screenshot if a test is not passing.

Screenshots saved with **App.Screenshot()** are located in your test project's directory: MyTestProject"\bin\Debug folder

```
app.Screenshot(“MyScreenshot");
```

Useful to track broken UI from failed tests or for quality reports.



Writing the test in **natural language** and **performing them manually** before coding them can help you write complete automated tests.

Running your tests

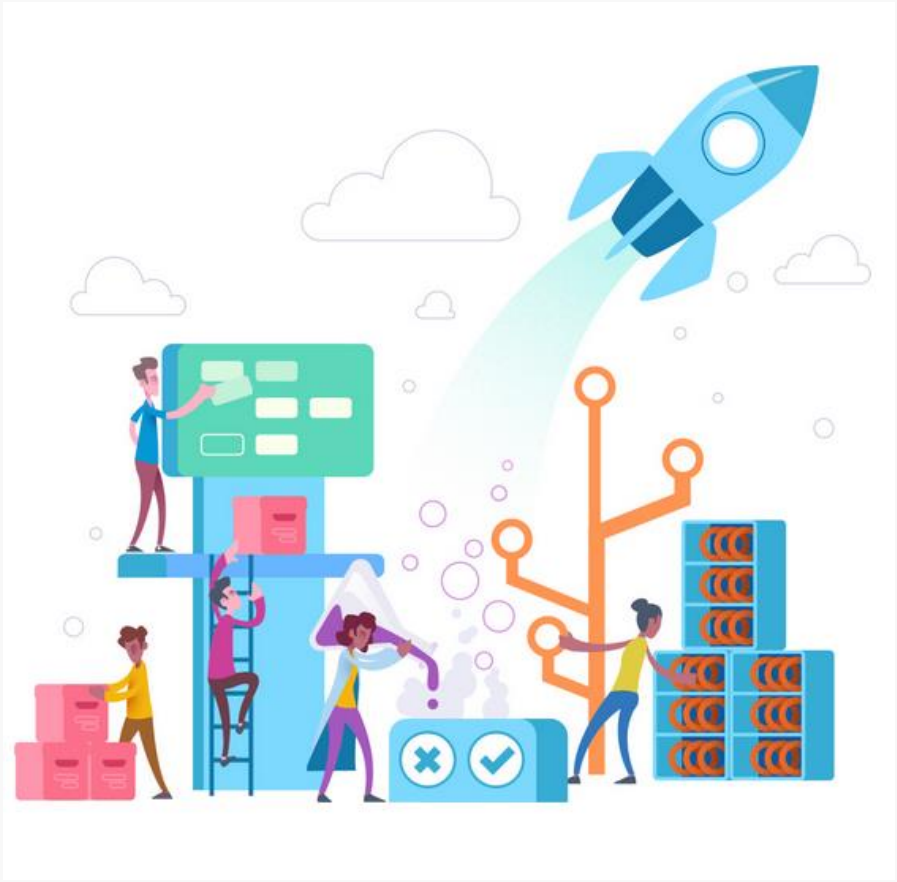
.NET

Add your Unit Tests to your DevOps pipeline

Get sources
codrinamerigo/ladybug branch1

Agent job 1
Run on agent

- Use NuGet 4.4.1
  NuGet tool installer
- NuGet restore
  NuGet
- Build Xamarin.Android project **/*Android*.csproj
  Xamarin.Android
- Build solution **/UnitTest.csproj
  MSBuild
- VsTest - testAssemblies
  Visual Studio Test
- Publish Test Results **/TEST-*.xml
  Publish Test Results
- Build solution **/UITest.csproj
  MSBuild
- Signing and aligning APK file(s) $(build.binariesdirector...
  Android signing
- Publish Artifact: drop
  Publish build artifacts
- Test with Visual Studio App Center
  App Center test
- Deploy $(build.binariesdirectory)/$(BuildConfiguration)/...
  App Center distribute



DevOps Test Reports

Tests in the pipeline

99.77%
Pass rate

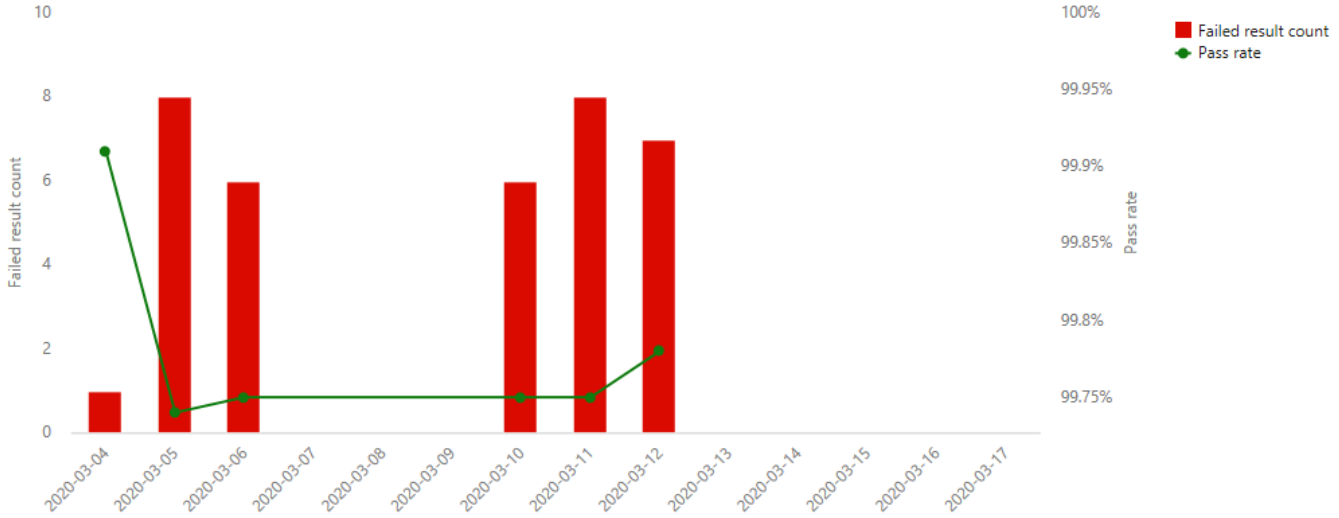
16K
Test results

16K Passed
36 Failed

Unique failing tests

3 tests causing 36 failed test results

Trend of test results and pass rate



Test	Failed ↓	Pass rate	Total count	Average duration
Get_Locations	18	0%	18	1.73s
Check_Status	17	5.55%	18	0.12s
Check_Status	1	94.44%	18	0.02s

https://dev.azure.com/xamarin/public/_test/analytics?definitionId=7&contextType=build

Run your UI tests on App Center Test Cloud

New test run

Android Test

4 configurations




Configure your run by selecting

Test framework:

- Appium
- Calabash
- Espresso
- Xamarin.UITest

Upload package and UI Test using ***nodeJs*** and following app center instruction

appCenter Validation

Wait for devices

Run on first device when ready

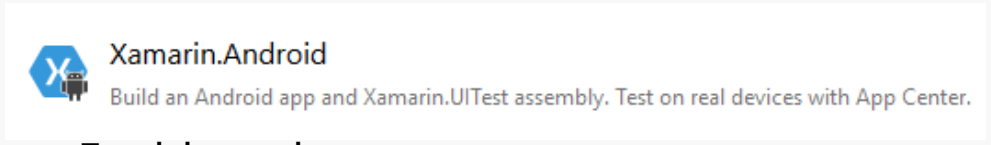
Run on second device when ready

Run on third device when ready

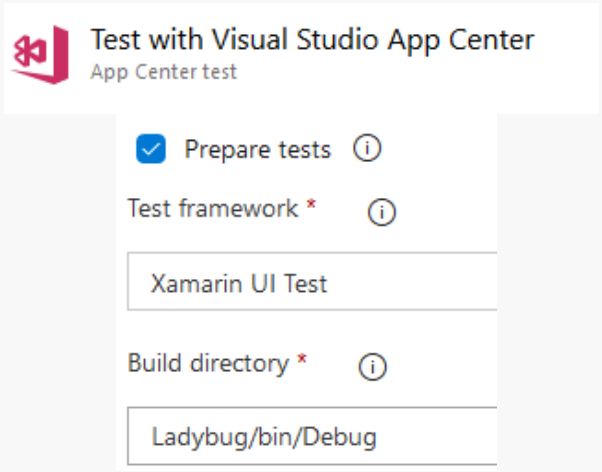
Run on nth device when ready

Generate report for results

Add UI Tests to DevOps using AppCenter

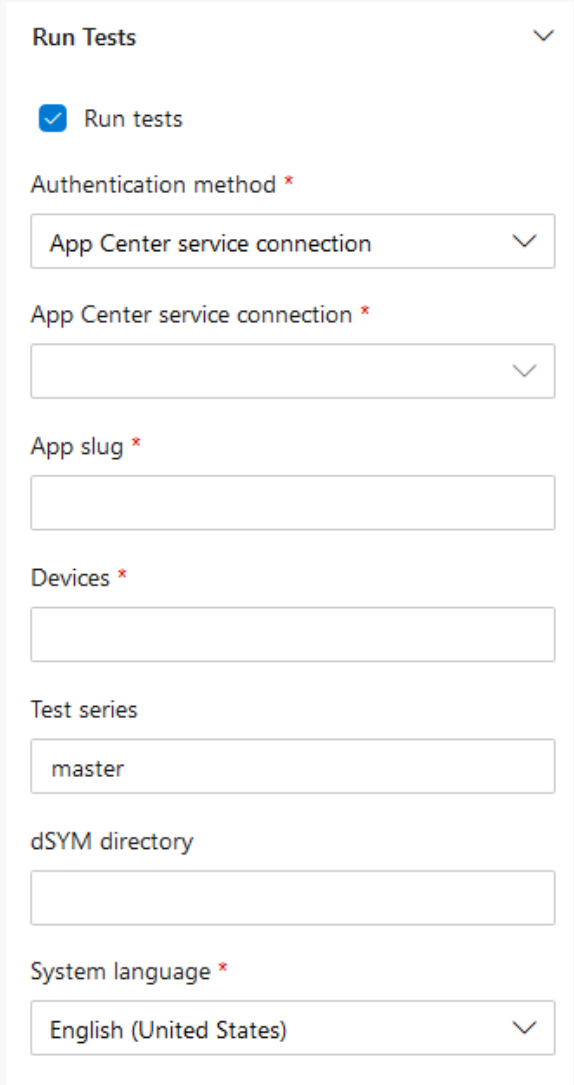


Enable task

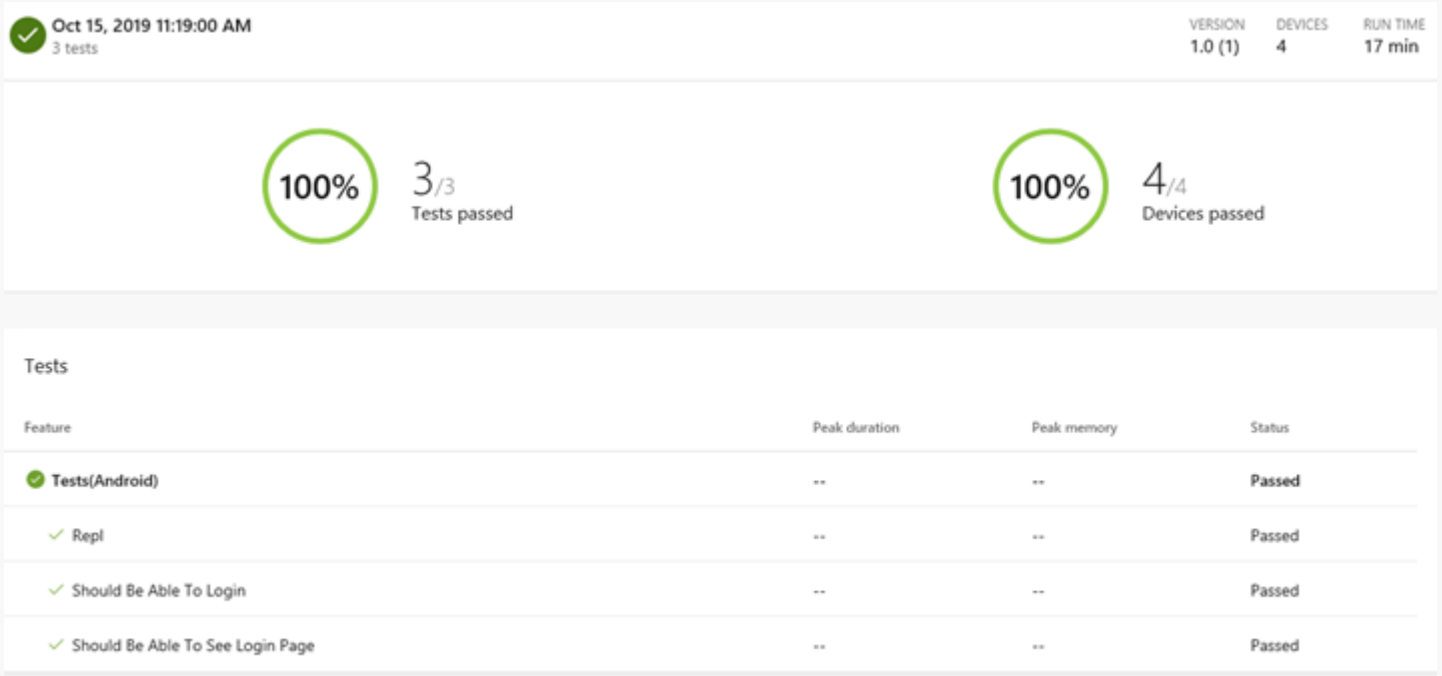


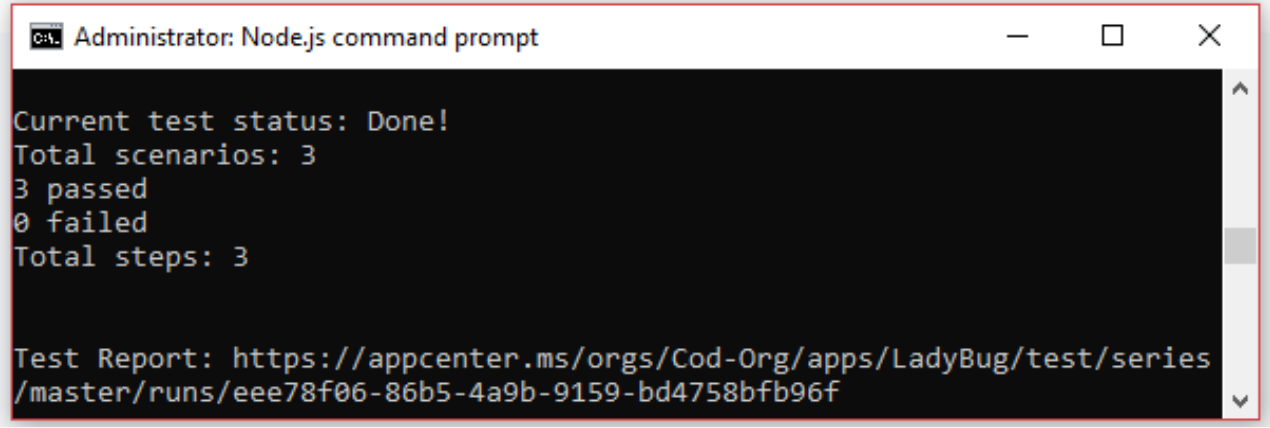
From app center you get:

appcenter test run uitest
--app "codrinamerigo/LadyBug"
--devices "codrinamerigo/android-test"
--app-path pathToFile.apk
--test-series "master"
--locale "en_US"
--build-dir pathToUITestBuildDir



App Center Test Cloud Reports





Keep your tests alive!

Q & A

.NET

Thank you

@_Codrina_