6

Implementing Automatic Testing

using Xamarin

In today’s world of fast paced development and frequent distribution, the application needs to be delivered as fast as possible and the development life cycle must be reduced to meet this goal of fast delivery. Testing is one of the most important aspects of software development.  
In case of mobile applications, it needs to be tested on all possible supported devices to make sure it delivers smoothly on all the targeted devices.

**Understanding the importance of automation testing in DevOps Cycle**

Testing is one of the most important factors when it comes to application development. Developers do their best to develop a sophisticated application that runs smoothly. But there are always scenarios that developers cannot think of just because they can only be tested when application is used with the mindset of an end user.

While developers are working hard to develop the application it is very crucial for the application to be tested with the user in perspective and to test things a developer wouldn’t think of.

Testers are there to make sure the application performs as it’s intended behavior and one feature does not affect the other features in an application.

While it is great to have manual testers testing the application and it does make sure that the application is tested and used as any end user would actually use it in the real world. But it is not always the best choice to only have manual testers test the application.

**Testing a mobile application**

When it is a web application or a desktop application, the number of platform versions and devices are very few to test with.  
It is always simpler to have manual testers test the application and find out the shortcomings and defects in an application and notify them to developers.

But when it comes to mobile application, the story completely changes.  
if we only talk about an Android application for example, let’s have a look at the number of challenges in testing.

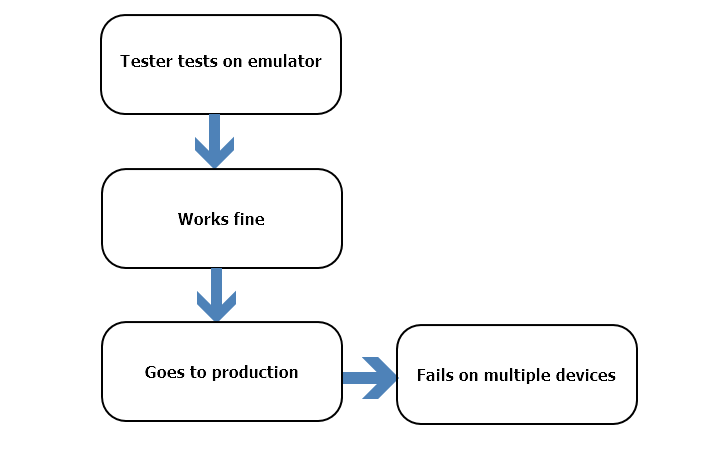
**Challenges in testing a mobile application**

There are many challenges when it comes to testing a mobile app

* Test against real environment
* Deploy and test frequently
* Continuous feedback

**Testing against real environment**

The most important thing for mobile developers is that the final app works across on all its target devices. Using emulators or simulators are fine in the earliest phase of the development, but when app becomes more sophisticated and about to get released to the market, the only acceptable way is to test mobile apps on real devices.

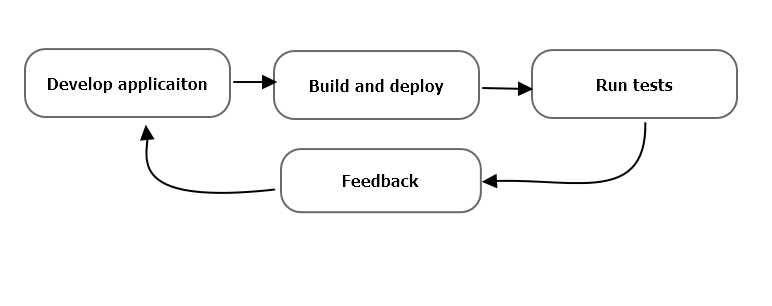


**Deploy and test frequently**

Mobile applications are updated almost every week or at least twice in a month. So they need to be tested even more frequently.  
Mobile applications should be tested with every nightly build so the defects can be recognized earlier and fixed sooner. Testing the same feature again and again efficiently and frequently might not be the best solution and can slow down the process of delivery. If we have automation in place for repeated tasks, that can save a lot of time to market. Continuous development and testing enables companies to deliver to market needs faster and in timely manner.

**Continuous feedback**

Continuous feedback goes along with frequently deploying and testing the application. As we have learnt that manual process of repetitive tasks makes us spend lot of our time and can be and should be saved in order to deliver to market faster. In the same way getting continuous feedback from testing and production is very important to the quality of application.



To overcome all the above challenges in mobile application testing we need to adopt DevOps mindset and make testing an automated and integrated part of our development cycle that can happen on each build automatically and give feedback to developers to take action on the early stages of development.

**Using Xamarin Test Cloud to Test on Multiple Devices**

Mobile application users are very demanding in terms of quality and performance of an application. And thanks to platforms like app store where users can promptly give their feedback as they wish makes it even more important to take mobile application quality seriously.

Q. How can we test a mobile application effectively?

A. By running the application on a real device and using like a user would do.

**Challenges:**

1. There are **various versions of mobile operating systems** people use
2. There are various devices out there with **different screen sizes**

**1. Different Mobile OS Versions**

Let’s take Android for example, since the time Android started it has grown with a very fast rate having a new major release every year. That makes almost 15 major versions out there in the market. Out of which at least 6 versions are widely used in different parts of the world depending on the region and smartphone availability in current date.  
This is only the major versions we are talking about and minor versions are way too many to count.  
Considering all these situations, it is almost impossible for a quality assurance team to test the application and guarantee to work on all the targeted devices out there.

**2. Devices with Different Screen Sizes**

We are all aware of the number of devices increasing every day with different screen sizes but old devices still continued to be used. From 4.0 inches to 6.5 inches and some even bigger.  
Mobile application can behave differently on different screen sizes and resolutions. If not developed properly it can drastically change its look from one device used for development to the one real user using.  
So testing on multiple devices with varied screen sizes becomes a very important aspect of mobile app testing. To an extent this can be done with getting all possible device sizes and testing application on them, but in turn can increase the cost to a very high level and can be a lot of time consuming.

To solve challenges like stated above in a cost and quality effective way is only possible by using cloud test platforms that enable us to run tests on multiple devices simultaneously. All the devices on these cloud platforms are physical devices and not emulators and they also provide instant feedback and support multiple testing frameworks like NUnit.

**Introduction to Xamarin Test Cloud**

Xamarin Test Cloud is a cloud-based platform that provides tools to support automated Testing of mobile applications across various different devices also known as UI Acceptance Testing. This enables us to ensure that the application performs correctly and efficiently across multiple devices with minimal effort.  
It also helps shifting the testers focus from repeating the same tests on multiple devices and helps them focus on verifying that app works as expected on the test cloud.

The Xamarin Test Cloud family consists of the following parts:

* Xamarin.UITest
* Test Cloud
* Xamarin Test Recorder

**Xamarin.UITest**

It is a testing framework based on very popular NUnit test libraries. If you have used NUnit in your C# projects earlier for unit testing it can be really helpful to understand Xamarin.UITest faster. But if you don’t have prior experience in NUnit then it’s absolutely fine.  
It is basically a set of libraries for C# (similar to JUnit for Java) to help write unit tests.  
Xamarin also supports Calabash framework for writing tests if you want to write them in Ruby and Cucumber.  
We’ll be focusing on Xamarin.UITest for writing tests in C# for continuous testing.

**Test Cloud**

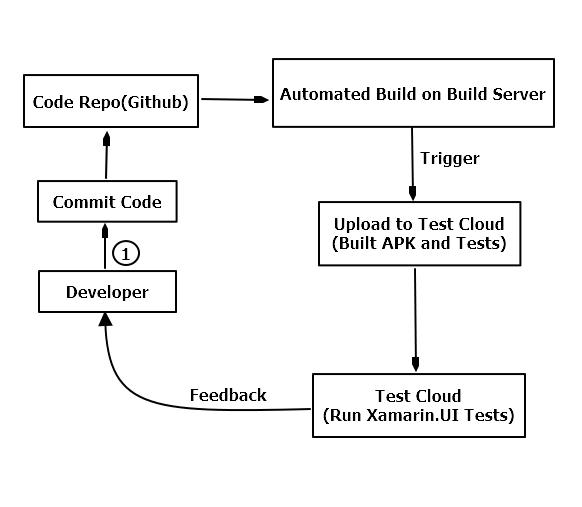
It is a cloud based platform consisting of thousands of physical devices. Users can upload apps and tests written in Xamarin.UITest to Test Cloud, it will then install the apps on the available or chosen devices and run the given tests on them. Once tests are complete, results are then available to users to analyze and verify the behavior of the application.

**Xamarin Test Recorder**

It is another application in the Test Cloud ecosystem that helps with writing Xamarin.UITest.  
It basically allows you to plug the device in, run the test manually on the device and write all the test code for you by recording your actions on the application.  
We will not be covering Test Recorder and rather learn how to write UI tests with Xamarin.

**Using Xamarin Test Cloud as Part of Continuous Integration**

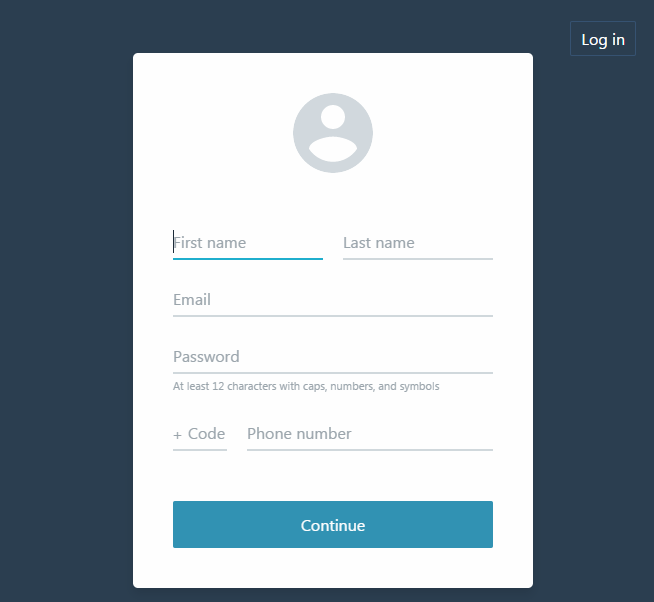
Xamarin Test Cloud helps us achieve continuous integration with automated test executions on every build on a build server like TeamCity and after executing the tests, giving feedback directly to developers.



**Creating Users and Organization on Test Cloud**

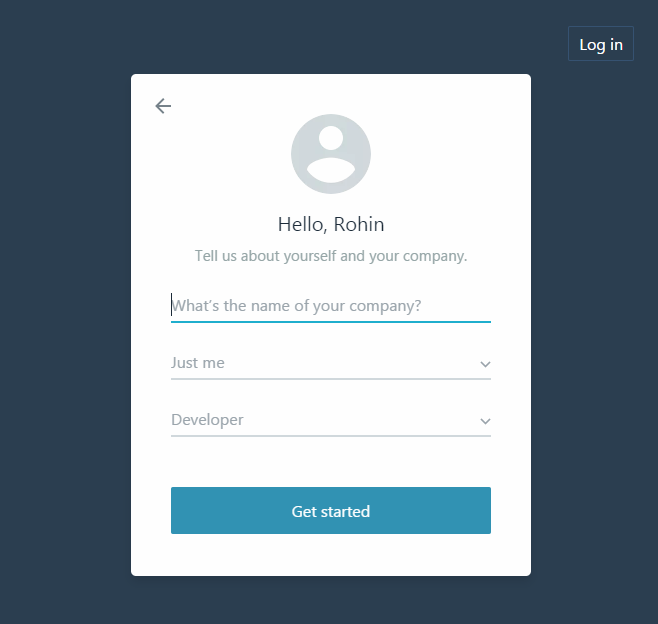
Let’s start by creating an account on Xamarin Test Cloud.

1. Go to <https://testcloud.xamarin.com/register> to register a new account.

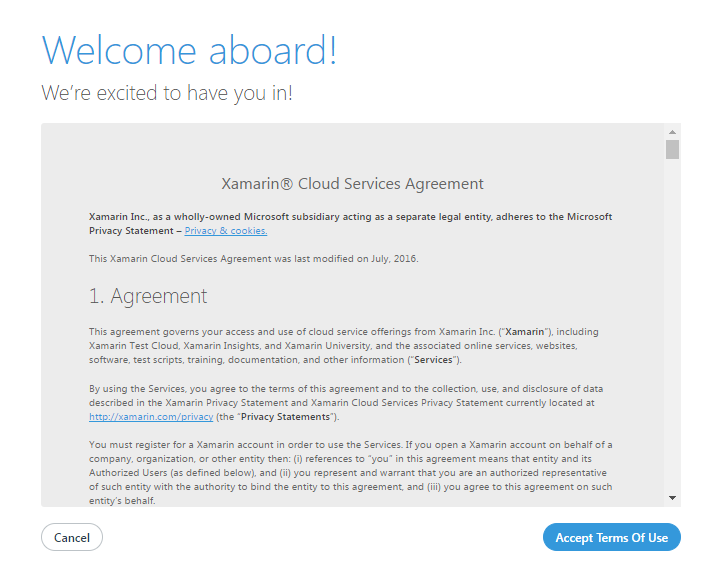


Enter your details and click “Continue” button to start the process, the process requires you to register with a company email, a Gmail or Yahoo email would not work.

1. On the next screen enter your organization details and click get started.



1. On the next step, accept the terms and condition to complete registration



1. After completing the registration, make sure to verify your email address before beginning testing.

**Users and Organizations**

Xamarin Test Cloud being a continuous testing cloud platform, supports an organization structure to give access and make APIs and run tests through some access keys.

It makes it much better to separate team based applications in an organization.

**Test Cloud Hierarchy**

The Test Cloud hierarchy structure is quite simple and easy to follow

1. **Organization**

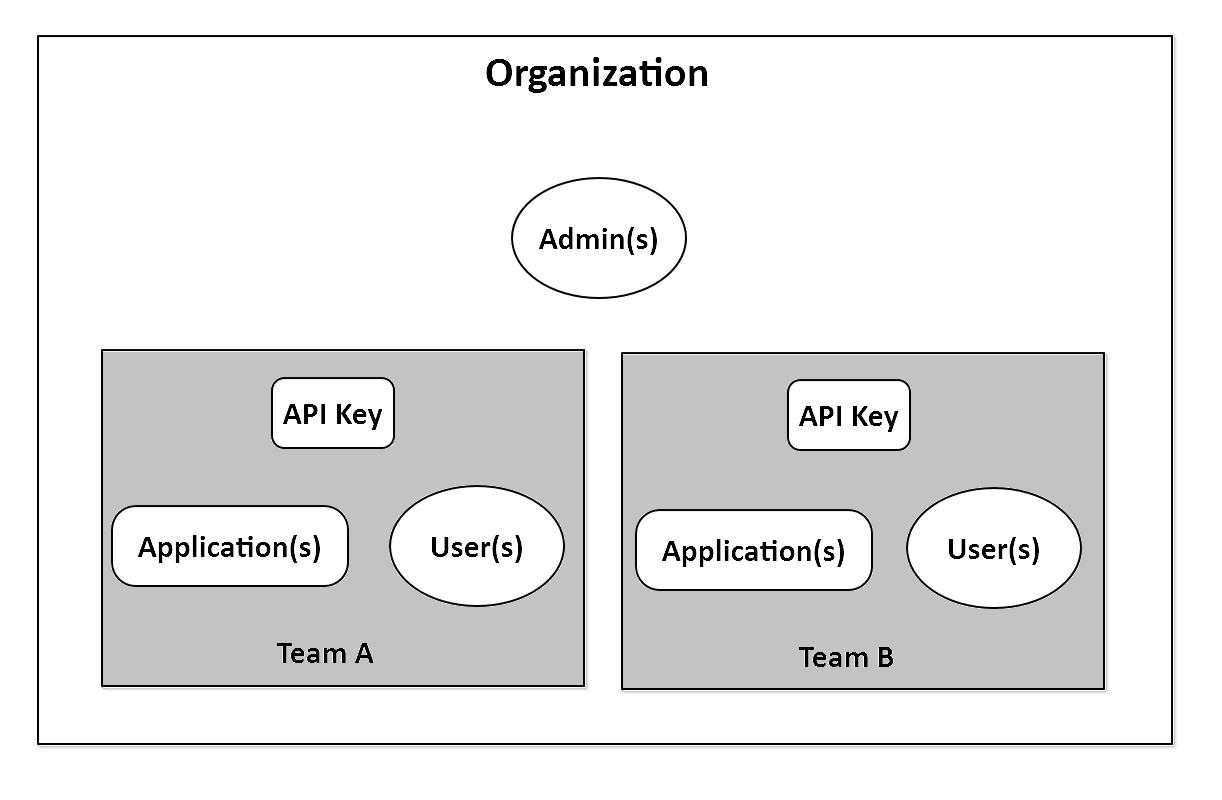
An organization is basically the top level at which the subscription is managed for the Test Cloud and it is created when a person from organization first creates an account on Xamarin Test Cloud.

1. **Administrators**

Each organization would have at least one administrator who creates teams and can manage users

1. **Team**

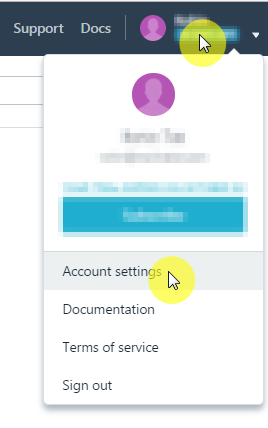
A team usually has at least one application and some users working on that application. Each team gets their own API keys to access and run tests on the application.



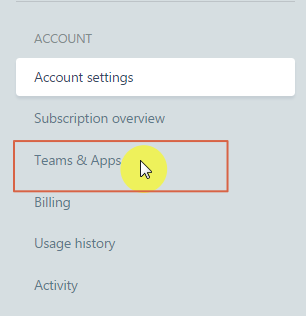
**Creating a Team**

To create a team in Xamarin Test Cloud, follow below steps.

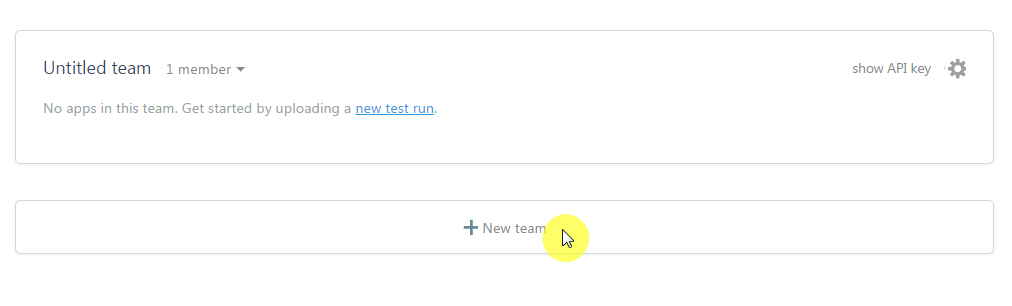
1. Click on Profile and then Account Settings



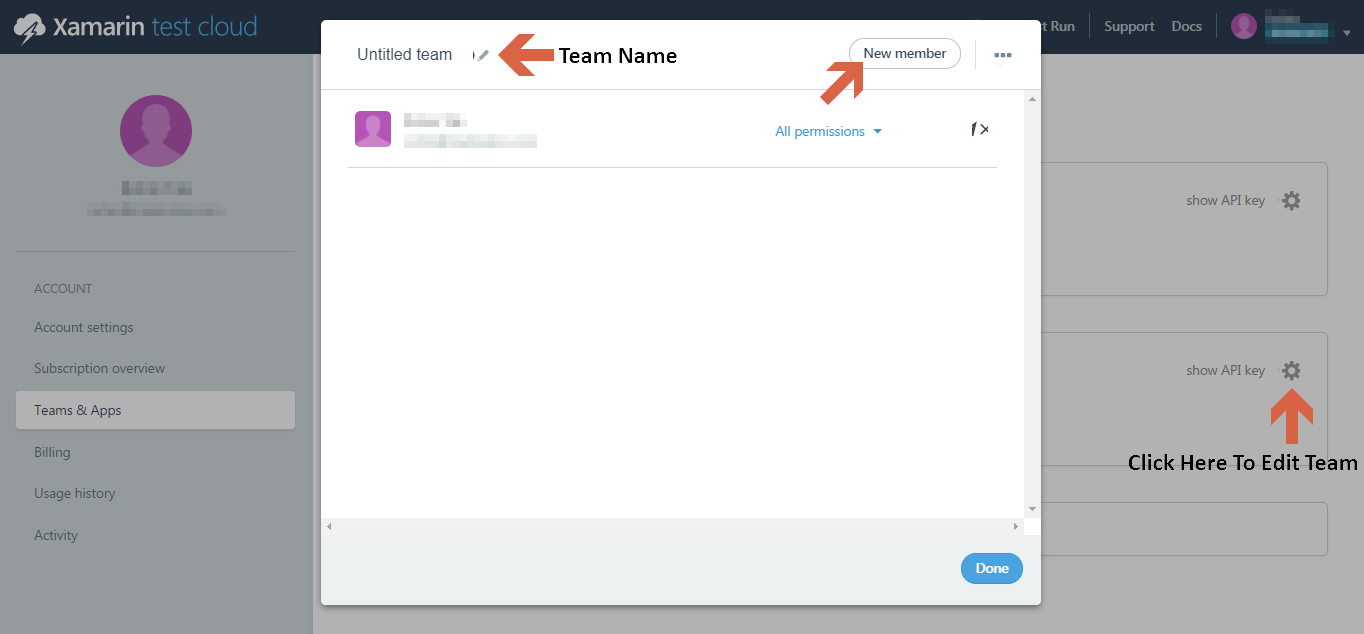
1. Click on Teams and Apps



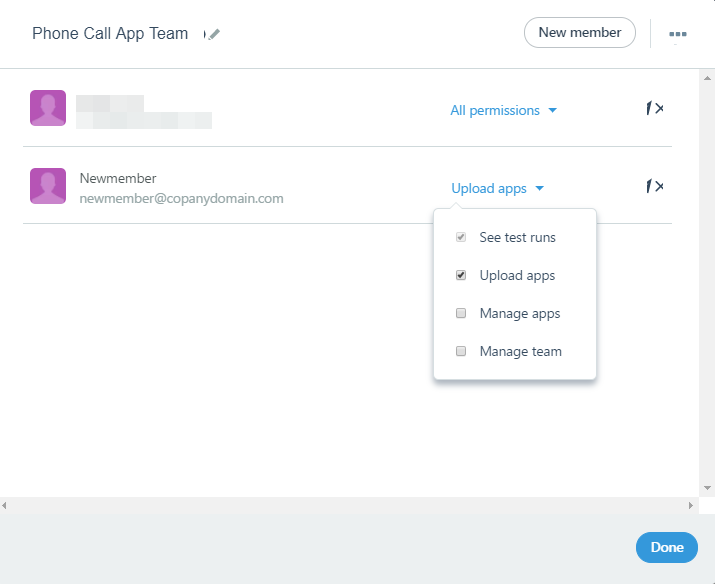
1. Then click on New Team button to add a new team



1. To edit team details such as team name and add new members, click on the settings (gear) icon as shown below



1. After adding new members to the team, you can manage permissions for the user



1. Once the team details and members and their permissions are all setup, click on Done.

**Writing tests with Xamarin.UITest**

Before we start writing UI tests with Xamarin.UITest, it is good to get a brief knowledge about what unit testing is and how Xamarin.UITest is going to help us achieve our goal of automated testing for our mobile application.

**Xamarin.UITest**

Xamarin.UITest is a testing framework based on popular test library in C#, NUnit. If you have used NUnit in your C# projects earlier for unit testing it can be really helpful to understand Xamarin.UITest faster. But if you don’t have prior experience in NUnit then it’s absolutely fine.  
It is basically a set of libraries for C# (similar to JUnit for Java) to help write unit tests.  
And by using UITest we will be writing UI Tests for our mobile application.

**Fundamentals of UITest**

* **Test Fixture**

Test fixture is a class containing tests and it also does the initial setup for any test to be executed or any task that needs to be done after the test is done executing

* **Test**

UITest is written as a method inside the Test Fixture class.

**AAA Pattern**

AAA pattern, also known as Arrange-Act-Assert pattern should be followed when writing a UITest to achieve the best results and fast feedback from the Test. As the name suggests, it consists of 3 steps:

1. **Arrange**

It can be judged by the name that this step contains all the actions that help arranging the test.  
E.g. initialization of things required later while running the test, setting up environment etc.

1. **Act**

This is when the test performs the desired interaction with the application like entering a text, pushing button etc.

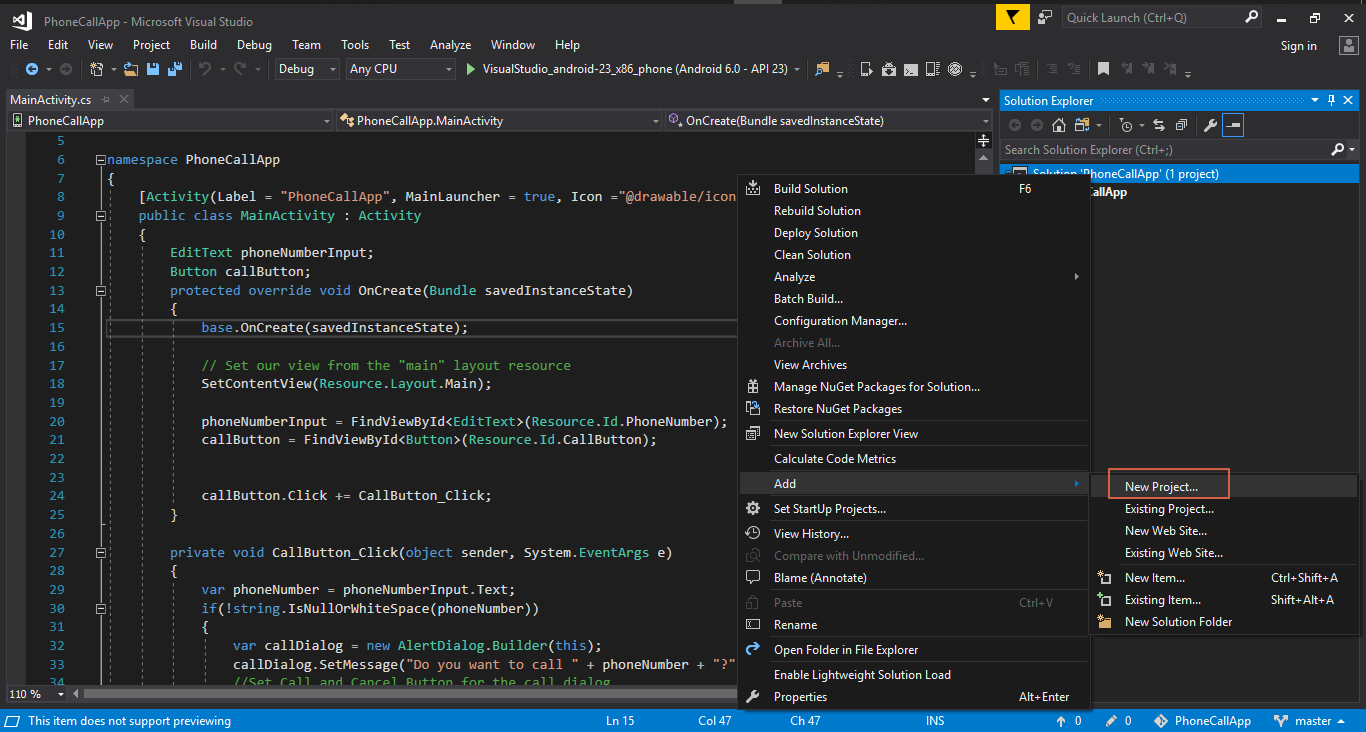
1. **Assert**

Assert is when our UITest asserts whether the interaction gave us desired outcome or not, like verifying that an error message was displayed or not.

**Add UITest Project to the Solution**

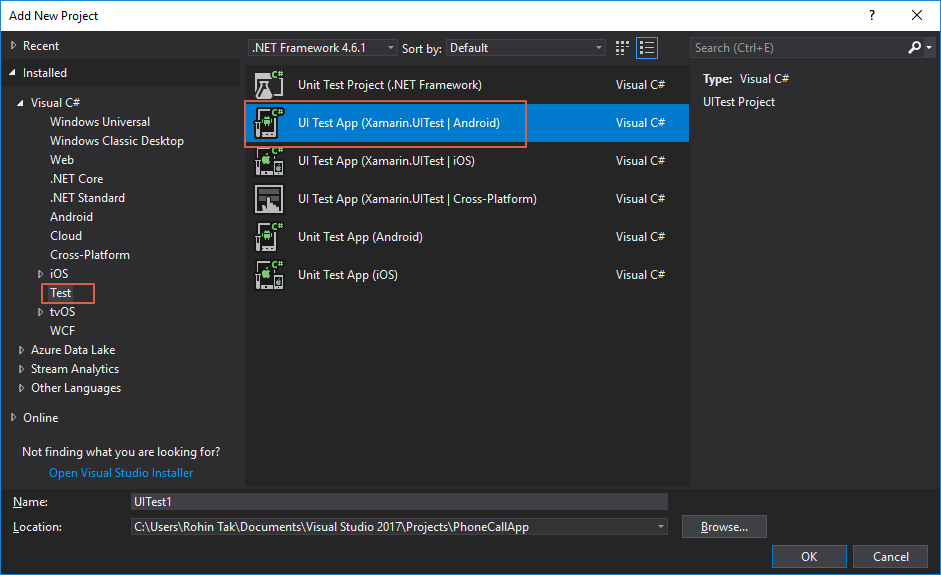
Let’s get back to Visual Studio and setup a new test project for writing Xamarin.UITest.

1. Right click on Solution -> Add -> New Project

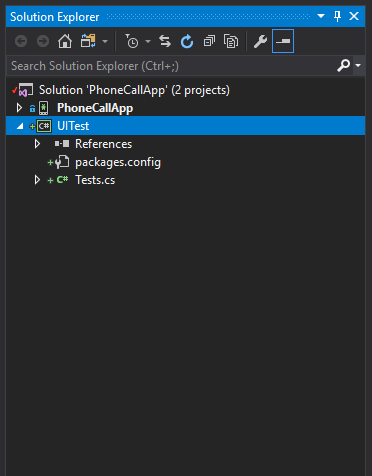


1. In the Add New Project windows, click on “Test” from the left section and then select “UI Test App (Xamarin.UITest | Android)”, because we are going to write test for Android application.

Give this project a name in the Name section and click OK.



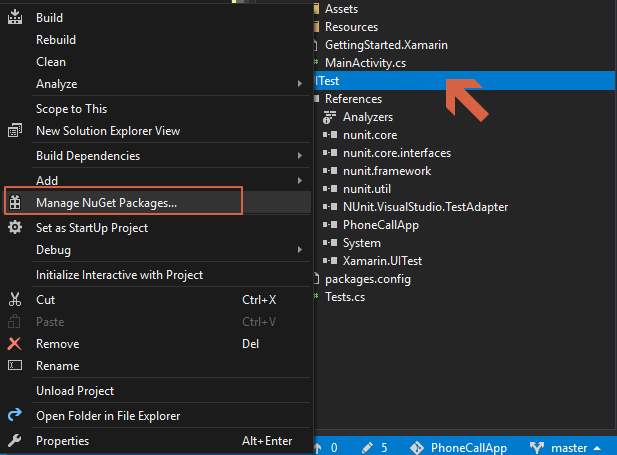
1. Visual studio will create a new test project now, once it is done you should be able to see a new project create under the solution



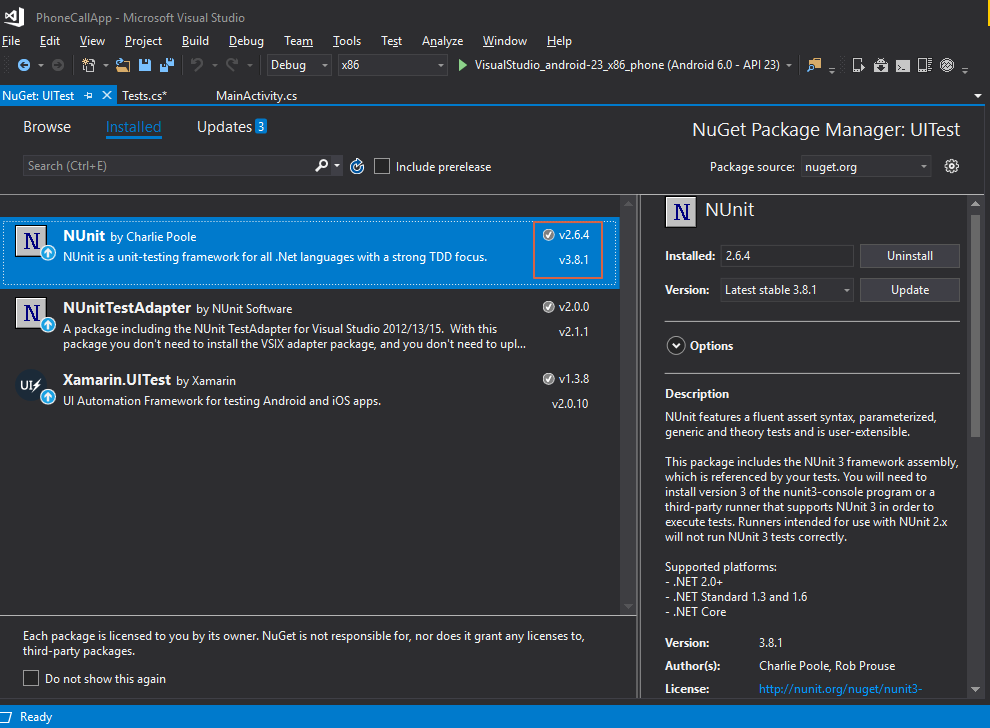
1. This new test project that we created through Android Test Project template should also include the necessary Nuget packages required to run Xamarin.UITest.  
   Those packages are:
   1. NUnit
   2. NUnit Test Adapter, to run UITests in local
   3. Xamarin.UITest, the framework we’ll use to write tests

These are available in the form of Nuget packages, but usually come with the template we used to create UITest project.

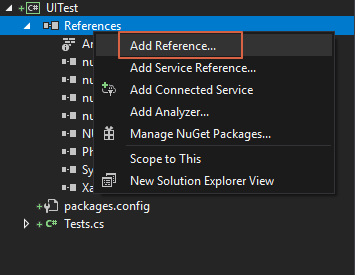
1. To make sure these required packages are available and installed, right click on the UITest project and click on Manage Nuget Packages



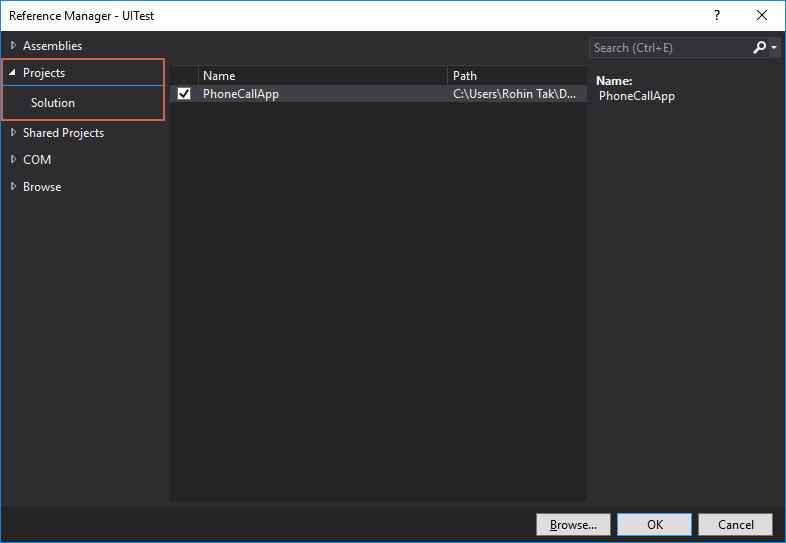
1. On the next screen you should be able to see the packages listed.



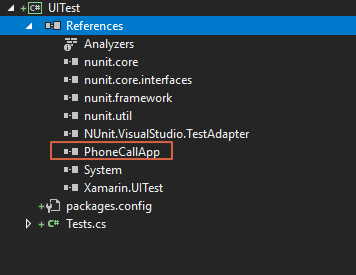
1. Visual studio suggests update for the NUnit package to 3.X.X, but do not update NUnit.  
   Because Xamarin.UITest does not work with NUnit 3.x, As of now it is compatible with 2.6.x.  
   Also because a version of test adapter is specific for a version of NUnit framework, it’s better not to update Test Adapter either.
2. Next we need to add a reference to the application project, so UITest project can build and run the application.
3. Right click on References under UITest project and click on Add Reference.



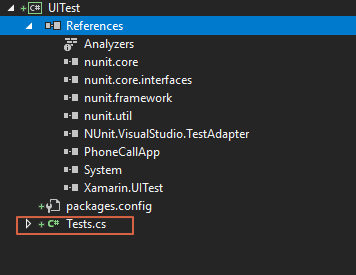
1. On the next screen, select Projects from the left section and then select the PhoneCallApp (the application project we want to test) and click OK.



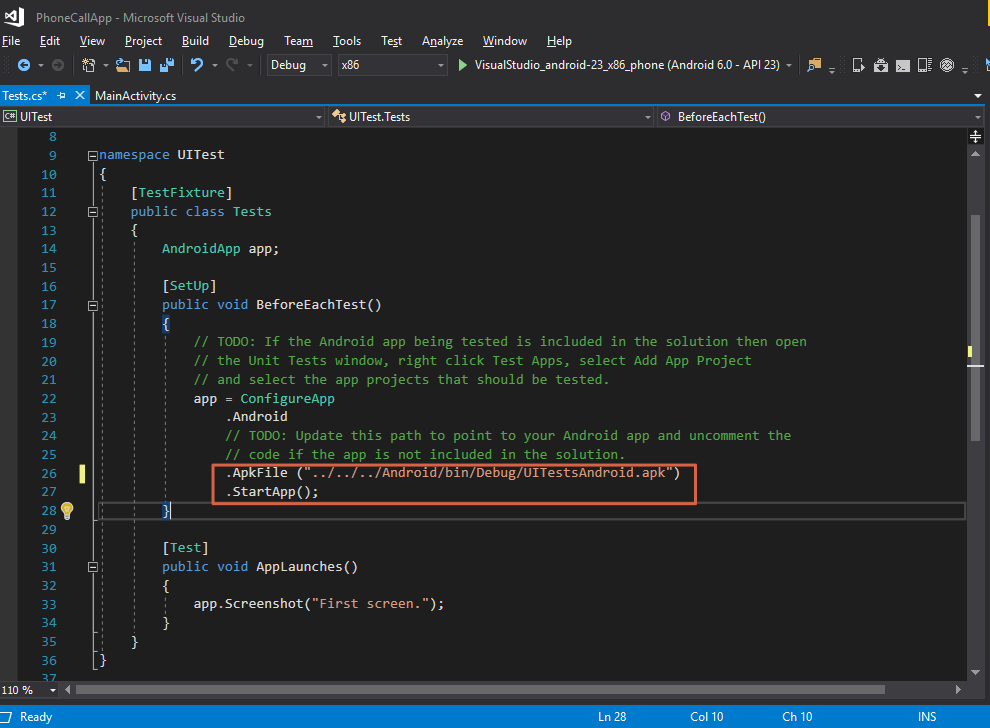
1. Once you add the application project, you should be able to see the reference added in the solution explorer.



1. Because the application project and the UITest project is under the same solution, it is enough to add reference to application project as shown in the previous step.  
   But **if you want to have both projects in different solutions or if you want to test application on Android 6.0** then Visual Studio requires you to give path to the APK in your system.
2. To give this path, open the Tests.cs file under UITest project.



1. In the Tests.cs, uncomment the “.ApkFile()” code as shown in the image below



1. Change the path to the .apk file path which can be found inside bin folder of your application project.  
   If you are not able to see the file there, try deploying application once and then it should be created in the bin/debug or bin/release depending on you build configuration.

**Writing Test for PhoneCallApp**

Everything is now setup and we are ready to start writing UITests for PhoneCallApp.

**Tests.cs**

This file is the default file that gets created when we add the UITest project in the solution through the Xamarin.UI Android Test project template.  
There are certain things to note in this file:

1. **[TextFixture]**

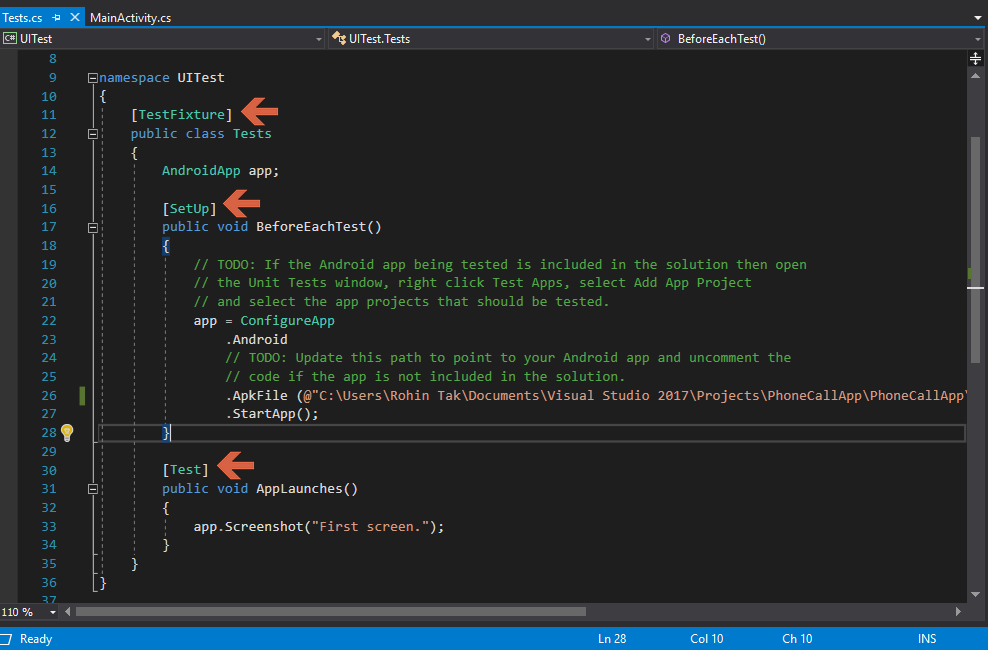
This is an annotation added to the Tests class that tells the UITest framework that this class contains tests to be run.

1. **[Setup]**

Each class containing tests need to setup initial configuration, like we added the APK file path in the previous section, this is added to the method BeforeEachTest() and tells the framework to run this method and setup initial things before running test

1. **[Test]**

This annotation identifies the method that contains the test



Let’s rewind some of things we wrote during the application development.

**Elements in the PhoneCallApp:**

* Textbox to enter phone number
* CallButton to call
* A toast that gets displayed if user taps on CallButton without entering a number
* A confirmation dialog box that comes when user taps on CallButton after entering a number

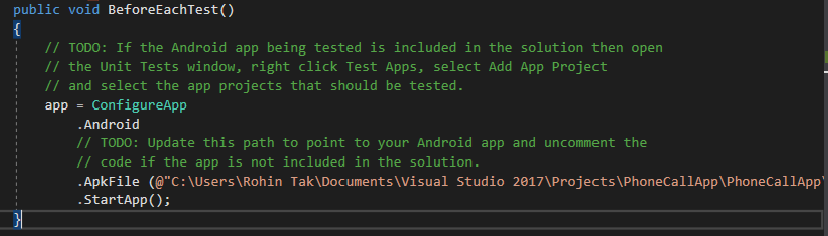
**User interactions in the PhoneCallApp:**

* Enter a number in the text box
* Tap the Call Button
* Tap on OK or Cancel on the confirmation dialog box

**Steps to include in the Test**

1. **Configure and start the application (Arrange)**

We need not write this step because that part is already taken care in BeforeEachTest() method.



1. **Perform an interaction with some element on the screen (Act)**

We’ll have to write code to enter text or tap on the CallButton

1. **Verify the desired output**

Code to verify if the interaction gives us desired output.

**Understanding the Test Case**

For writing a test, it is good practice to first define a test case to have a clear idea.

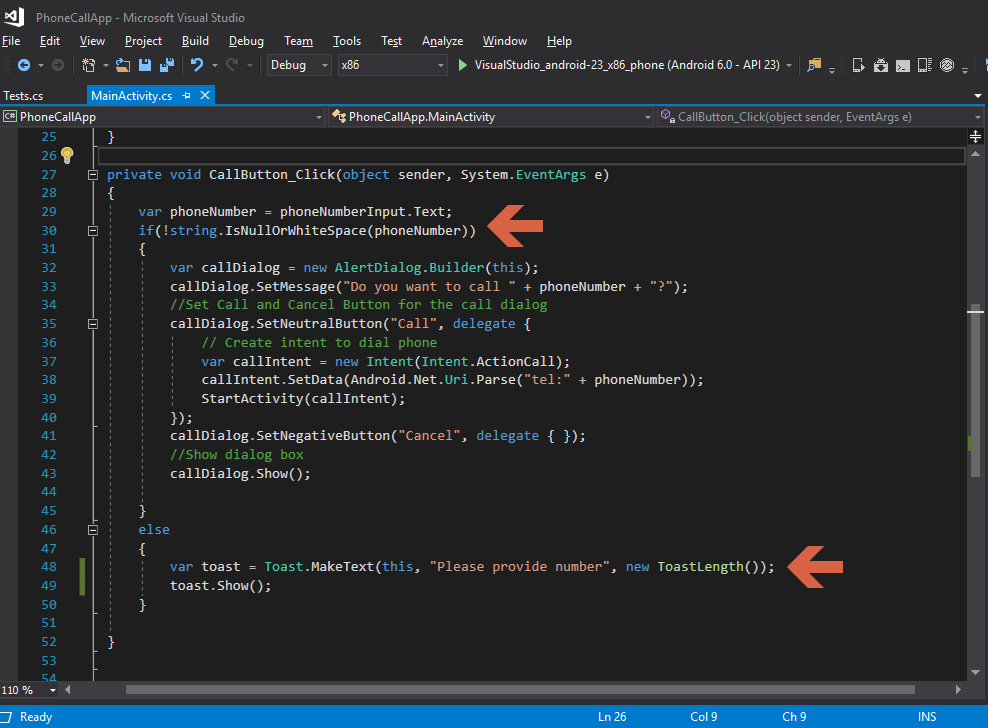
**Test Case:**

Click on CallButton (Id given to the “Call” button)

**Desired Output**

A toast gets displayed saying “Please provide number”

Open the **MainActivity.cs** from PhoneCallApp project and see CallButton\_Click method where we handle the user interactions.



It’s time to finally write our new UITest inside **Tests.cs** under UITest project to Test the test case described above.

1. Write a new method under Tests.cs class with an annotation [Test] as described earlier.

