7

Configuring Team City For CICD with Xamarin

In the old days, application development used to happen in separate not so integrated teams and developers were not used to merge their work with other developers’ code for quite long time and that used to create merge issues. Things that were working once on the developer’s local copy, used to stop working when merged with others’ code. This kind of not so integrated development environment increases the development time and delayed discovery of issues. Solution for this is to have continuous integration build into the development cycle where developers are merging their code multiple times a day and getting issues fixed at earlier stages.  
In this chapter we will learn more about Continuous Integration and Continuous Delivery and different tools that we can use for the same for a better development integration and delivery process.

**Introduction To Continuous Integration**

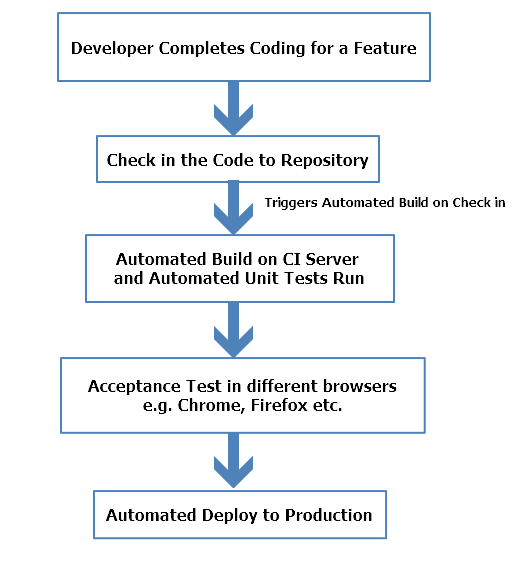
**Continuous Integration** (CI) is a development and integration practice in which developers checkin code into a shared repository frequently, preferably several times a day. Each code merge can then be verified by an automated build and automated tests if applicable.

There are many benefits of following continuous integration, one of the advantage is that it helps detect defects quickly and at early stage. The check ins are usually very small and contains small portions of developments, thus helping in identifying the exact issues quickly.

**Continuous Delivery** (CD)in the other hand is the process after integration and as the name suggests, it is to make sure that the code base checked in is deployable at any point of time. Each environment from test to production can and mostly do have different configuration. Continuous Delivery makes sure that all configurations are always ready for the deployment to any environment and that the code passes all the tests necessary for release.

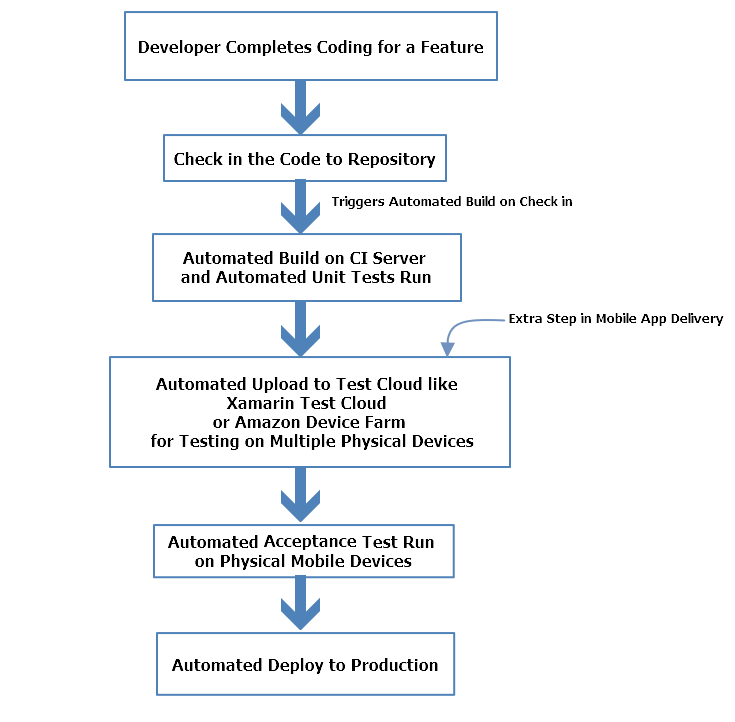
In short, Continuous integration improves the development and testing experience with frequent code merges and helps in quickly identifying the bugs and involves running automated tests if included in the process while Continuous delivery makes sure the codebase everything is in ready state for the code to be deployed in any environment.

**CICD for a Web Application**



For a web application once the build is ready from the CI server it is not a big task to test the application on different browsers since there are only limited number of them. But in a mobile application there is one more step involved to improve the experience since there are thousands of devices with different versions of operating systems available.

**CICD for a Mobile Application**



As shown in the image above, mobile applications need to be tested on hundreds of devices with different operating systems and purchasing on those mobile devices which keep on coming every now and then, can be very expensive. To make sure the quality of application stays at the top including Test Cloud based solutions becomes an integral part of the process.

**Choosing Tools for Continuous Integration**

There are many CI tools available in the market to implement continuous integration, just like there are many languages available in the market to developer applications. But choosing the right CI tool is very important for a good long-term benefit.

Choosing a CI tool for your project can depend on many variables like:

* **Programming Language Support**

This is one of the most important factors while choosing a CI tool. Some CI tools have better support for certain language specific builds and packages while others might not provide language specific packaging options.

* **Operating System**

Operating system support, meaning some teams might find an open source operating system like Linux to be a better choice for all their servers including the CI server and it might be more comfortable for them to configure a familiar operating system while other teams working on .Net applications might find Windows to be more comfortable and feature rich for there configurations. It all boils down to the preference different teams have and the kind of application they are working on.

* **Integration with Code Repository**

Different teams prefer different code repositories for various reasons, some might find Git to be more feature rich and supported on various IDEs with plugins while others who are more familiar with Microsoft environments find Team Services to be easier to use and better integrated. Different CI tools have different level of support for these repositories.

* **Support for Application Platform Deployment**

Some CI servers are better suited for web application deployment while other provide more features and better support for mobile application deployment to app stores. Depending on your type of application, the choice can vary.

* **Cost**

Cost is always an important factor while choosing any type of tool. Medium to big size companies can afford to have expensive feature rich CI tools while smaller companies and teams might want to stick to low budget and sometimes open source and freely available CI tools and customize them according to their needs.

**Various Tools for Continuous Integration**

Let’s have a look at some of the widely used CI tools available in the market.

**TeamCity**

TeamCity is a mature CI server, developed by JetBrains company. JetBrains is very popular in the software development world, and their tools like WebStorm and ReSharper are used by developers worldwide. It is a license based CI server but a free version of it is also available. TeamCity offers all the features in its free version, but it is limited to the 20 configurations and 3 build agents. Additional build agents and build configurations need to be purchased.

Despite being Java based solution, TeamCity **offers the best .NET support** among the tools on this list. There are also different enterprise packages, that scale by the number of agents needed.

You will be learning more about TeamCity later in the chapter.

**Key Features:**

* Great support for Visual Studio; versioning, testing, code coverage, code analysis, all available without any external scripts
* Detailed history reports for builds, failures, and any additional changes made
* TeamCity takes advantage of cloud computing by dynamically scaling out its build agents farm on Amazon EC2, Microsoft Azure, and VMware vSphere

**Jenkins**

Jenkins is one of the most popular open-source project for continuous integration. With thousands of plugins to choose from, Jenkins can help teams to automate any task that would otherwise put a time-consuming strain on your software team. Common uses include building projects, running tests, bug detection, code analysis, and project deployment.

**Key Features:**

* Jenkins works as a standalone CI server, or you can use it as a continuous delivery platform as well
* Pre-built packages for Unix, Windows, and OS X ensures an easy installation process
* A user-friendly web interface that can be used to quickly configure your server
* Custom plugins for build and source code management, administrative tasks, user interface, and platforms
* Large community with leading software brands involved in development

**Visual Studio Team Service**

Visual Studio Team Services provided by Microsoft helps teams plan better, code together, and ship faster. You can code in any IDE and language, for any target platform. Various tools and plugins can be downloaded to customize it to your project requirements.

**Key Features:**

* Kanban and scrum boards available for better planning and project management
* Unlimited Git and TFVC repos
* Hosted builds
* Automated release pipelines for better release planning
* Test and build commits before merging code to avoid build fails
* Built-in tasks and templates for setting up CI and CD to an Azure web app

**Bamboo**

Bamboo is a CI server being used by software teams worldwide to automate the process of release management for applications and general software, it allows teams to establish a streamlined pipeline of build delivery. Mobile developers can deploy their apps back to the Apple Store or Google Play automatically. Being the Atlassian tool, it has the native support for JIRA and BitBucket and you can even import your Jenkins configurations into the Bamboo easily.

**Key Features:**

* Bamboo can be used with Docker, AWS, and S3; it works out of the box with all of your favorite coding language
* Custom deployment projects to archive the history of each of your release version
* Detailed outline of your code history before you deploy, helping you understand the progress you’re making
* Compatible with Bitbucket and JIRA for a comprehensive CI experience
* With per-environment permissions, developers and QA can deploy to their own environments on demand while production stays locked down

**Using TeamCity with Xamarin for CICD**

As mentioned in previous section, TeamCity provides great support for .Net based applications.  
While it can automatically detect build steps from the configuration files and project files, it can also detect automatic build triggers from GitHub.

**Requirements for using TeamCity**

For using TeamCity, knowledge and availability of some hardware and technologies is required to make the setup process smoother.

1. **A dedicated build server for TeamCity Installation and setup**

Ideally the build server should be a standalone server and should not be responsible for other responsibilities like, DB server or hosting server etc.

1. **Knowledge of MSBuild**

Having knowledge of MSBuild can make this setup much better and would help in resolving any compilation related issues if required.

1. **Knowledge of Xamarin Test Cloud for Continuous Testing**

Xamarin Test Cloud will be used in this chapter for continuous testing after a build and application package is ready. You would be familiar with this because it has been described in detail in the last chapter.

**Steps Involved in TeamCity Setup**

There are several steps involved in setting up TeamCity

* **Preparing the Build Server** – This step involves installing the necessary software, tools, and certificates required to build mobile applications and prepare them for distribution.
* **Creating A Build Script** – Build scripts are used to perform different functions like compiling and application, packaging the application etc. Using a build script helps with troubleshooting build issues and provides a consistent, repeatable way to create the package for distribution.
* **Installing TeamCity** – As the name suggests, this step involves installing TeamCity on the build server for the first time, creating build agent and adding users.
* **Creating A TeamCity Project** – Once the previous three steps are completed, we must create a TeamCity project that will contain all the meta-data necessary to retrieve the source code, compile the projects, and submit the tests to Xamarin Test Cloud.

**Preparing the Build Server**

An important step in configuring a build server is installing the necessary tools, software, and certificates to build the mobile applications. It is required that the build server should be able to compile the application solution without any issue and should be able to submit any tests to Xamarin Test Cloud. To minimize configuration issues, the software and tools should be installed in the same user account that is hosting TeamCity.

1. Firewall Configuration
2. Installing Visual Studio with Xamarin
3. Installing Visual Studio SDK
4. Android Keystores

**Firewall Configuration**

For continuous testing, we are using Xamarin Test Cloud and it was described in the previous chapter in detail. For tests to be submitted to Xamarin Test Cloud automatically as part of CI, the computer submitting the tests must be able to communicate with the Test Cloud servers and firewalls must be configured to allow network traffic to and from the servers located at testcloud.xamarin.com on ports 80 and 443. This endpoint is managed by DNS and the IP address can change in future.

In some situations, a test or a device running the test needs to communicate with web servers protected by a firewall. In this scenario, the firewall must be configured to allow communication from the following IP addresses:

* 195.249.159.238
* 195.249.159.239

Once these configurations are done and firewall is configured to allow the communication between server and Xamarin Test Cloud, we will be able to use command line tools in the build steps to submit our UITests to Xamarin Test Cloud.

**Installing Visual Studio with Xamarin**

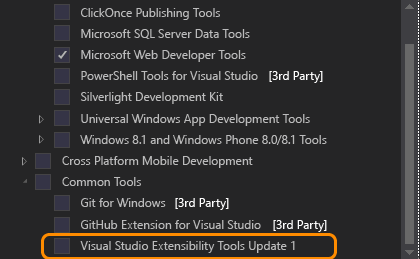
To install Visual Studio with Xamarin, you can follow the same steps described in 3rd chapter.  
It should install both Visual studio and Xamarin with required tools and SDKs.

**Installing Visual Studio SDK**

Visual Studio SDK can be installed while installing the Visual Studio Ide itself as part of optional component.

If you decide to install the Visual Studio SDK after completing your Visual Studio installation, you should follow the following steps:

1. Go to Control Panel / Programs / Programs and Features, and look for Visual Studio 2017.
2. Right-click Visual Studio 2017, and then click Change. You should see the installation page.
3. On the first screen, select Custom, not Default. Click Next.
4. You should see a tree view of custom features. Open Common Tools. You should see Visual Studio Extensibility Tools



1. Click the Visual Studio Extensibility Tools link to install the Visual Studio SDK.
2. Click Next and continue with the installation.

**Android Keystores**

Android Keystore is used for signing the application while distributing the application. This is required while before packaging the application so that our final package is singed with it.

**Creating your own keystore**

The first step is to create your own personal keystore that will contain the information used to digitally sign your Android package files. You can do this with the following command:

*"C:\Program Files (x86)\Java\jre1.8.0\_45\bin\keytool.exe" -genkey -v -keystore* *youFileName.keystore" -alias your\_alias\_for\_keystore -keyalg RSA -keysize 2048 -validity 30000*

The 30000 at the end of the command denote the length of validity of the certificates and Google require this to be past 2033.

Before you run this command make a note a few parameters first. When you run the command, it will ask you the following parameters to enter. These parameters will be used again later in the project file.

**Password** - <yourpassword>

**Name** - <yourname>

**OU** - <organisationunit> eg: JamSoft

**Orgname** - <organisationame>

**Local** - <locality>

**State** - <state>

**Country** - <2lettercountrycode>

Running the command should generate a .keystore file with the filename provided in the command.  
Now that our build server is ready, let’s prepare the build script that we’ll be using the build process.

**Creating Build Script**

The build script must be able to perform the following steps:

* **Compile the Application** – This includes signing the application with the correct provisioning profile.
* **Submit the Application to Xamarin Test Cloud** – This includes signing and zip aligning the APK with the appropriate keystore.

**Compile the Application**

Now that we have our android keystore ready and prepped for use we can look at the Visual Studio project. In order to make this automated in the build system we need to configure the project to use our keystore credentials.

1. In Visual Studio edit the Android application .csproj file and add another PropertyGroup element as per the code below:

*<PropertyGroup Condition="'$(Configuration)' == 'Release'">*

*<AndroidKeyStore>True</AndroidKeyStore>*

*<AndroidSigningKeyStore>myandroid.keystore</AndroidSigningKeyStore>*

*<AndroidSigningStorePass>yourpassword</AndroidSigningStorePass>*

*<AndroidSigningKeyAlias>myaliasdroidpub</AndroidSigningKeyAlias>*

*<AndroidSigningKeyPass>yourpassword</AndroidSigningKeyPass>*

*</PropertyGroup>*

1. Now our .csproj file knows how to use our keystore unattended. We can tie into the Xamarin build process from within our automated builds and produce the base Android package.  
   You can test that this is working using the following command:

*msbuild.exe PhoneCallApp.csproj /p:Configuration=Release /t:Rebuild*

This command uses MSBuild to build the application with the given configuration, in our case it should be release.

1. We have our application package now and we can apply the signing processes. To sign the package created in the previous step we need to execute the following command:

*"C:\Program Files (x86)\Java\jdk1.7.0\_71\bin\jarsigner.exe" -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore youFileName.keystore -storepass yourpassword -keypass yourpassword -signedjar \bin\Release\packagename-signed.apk \bin\Release\packagename.apk your\_alias\_for\_keystore*

This package is now digitally signed using your certificate from the keystore we made earlier.

1. Now that we have a signed package we can zip align this package and then publish this as an artifact of our TeamCity build process. This command makes use of the Android SDK zipalign.exe program. You’ll have to find where this is on your machine as there are many potential locations. The command you need will look something like this:

*“C:\Users\<name>\AppData\Local\Android\android-sdk\build-tools\<version>\zipalign.exe” -f -v 4 packagename-signed.apk packagename-zipaligned.apk*

1. Now it is time to upload our tests and android package to Xamarin Test Cloud to be UI tested. We had created Xamarin.UITest In previous chapter and it is assumed that you are aware of the process of creating and uploading the test to Xamarin test Cloud.
2. So include the following command to your build process to upload test to Test Cloud:

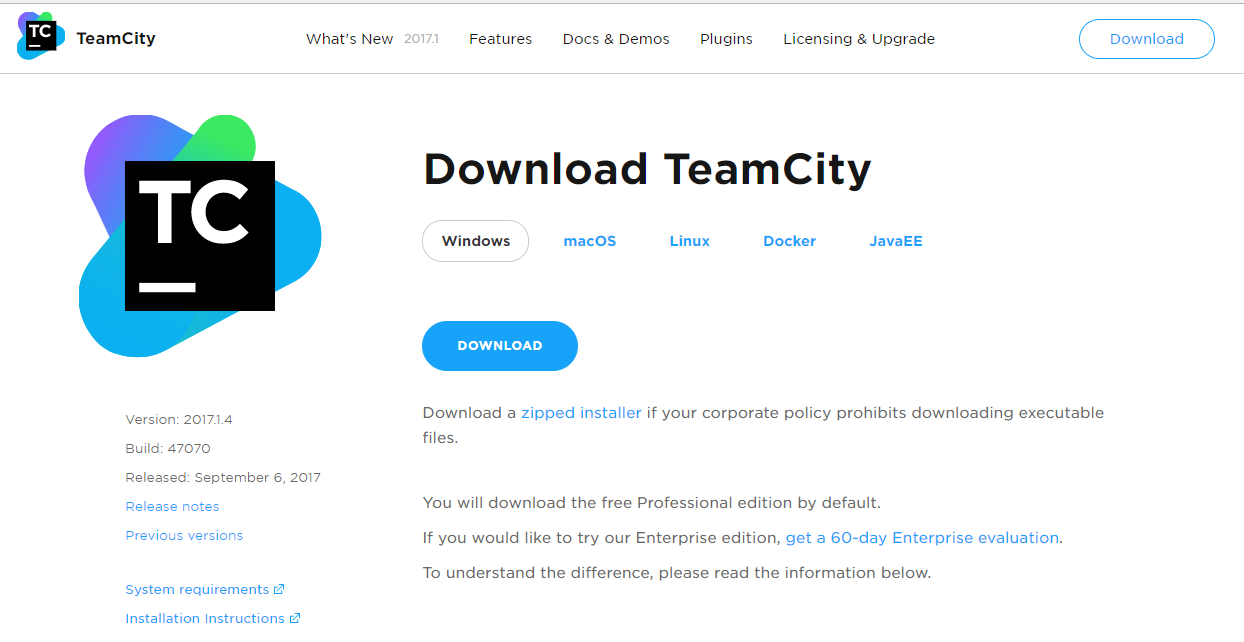
*test-cloud.exe <path-to-apk-or-ipa-file> <test-cloud-team-api-key> --devices <device-selection-id> --assembly-dir <path-to-tests-containing-test-assemblies> --nunit-xml report.xml --user <email>*

When the test is run, the test results will be returned in the form of an NUnit style XML file called report.xml. TeamCity will display the information in the Build Log.

**Installing and Configuring TeamCity**

To install and configure TeamCity on Windows machine, follow below steps.

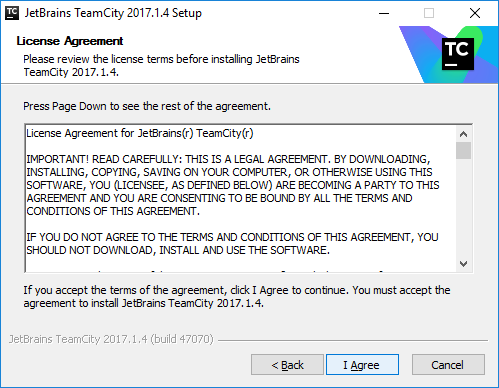
1. Go to URl <https://www.jetbrains.com/teamcity/download/#section=windows> and click on the download button to download TeamCity installation package from TeamCity website.



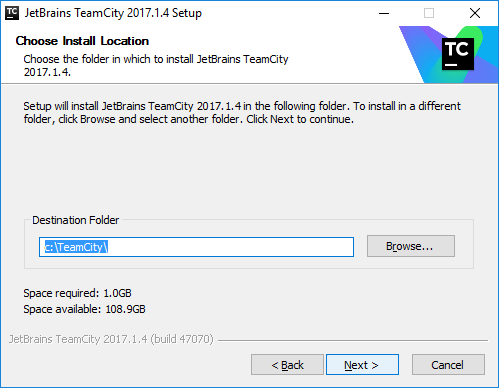
1. Once downloaded, open the installation package and click next.



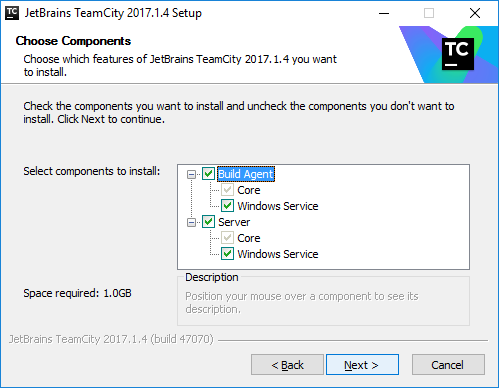
1. On the next screen, agree to the license and go to next step.



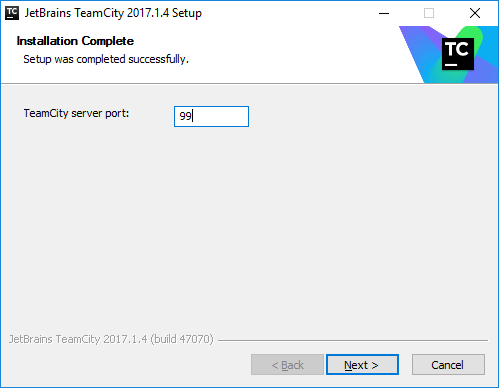
1. Select the path where to install TeamCity and click next.



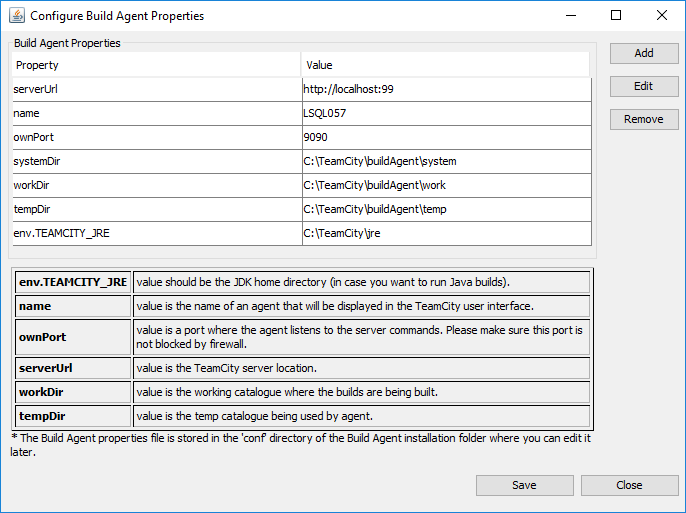
1. Select the packages to be installed, for example, if you are installing build agent and server on different servers then select accordingly. For learning purpose, you can select to install both on the same machine.



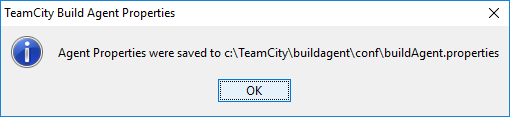
1. Once the installation is done, select the port where you would like TeamCity server to run on. Make sure this port is not used by other services on the machine and choose a unique port number and not the default one if possible.



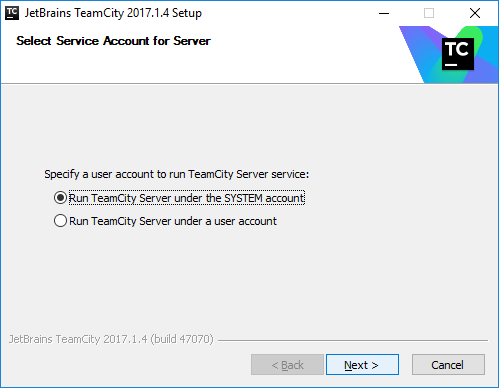
1. In the next screen, you’ll be able to see the configurations and ports configured for the server, you can also change them here if you want to.



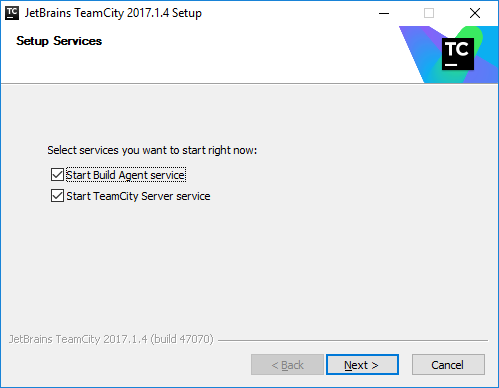
1. Click on save to save the configuration.



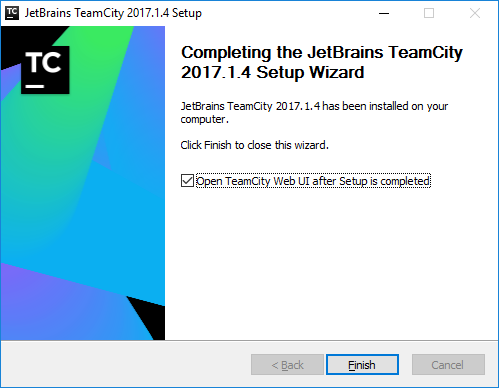
1. Select the account to run TeamCity.



1. Start Build server and build agent.



1. Check Open Team City Web UI and click finish.

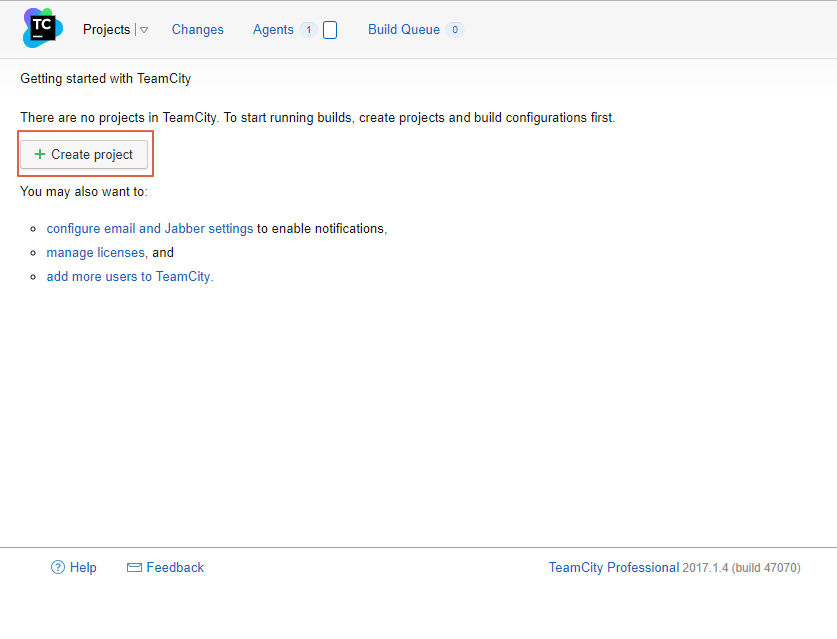


1. This will open TeamCity web UI where we can create TeamCity Project.

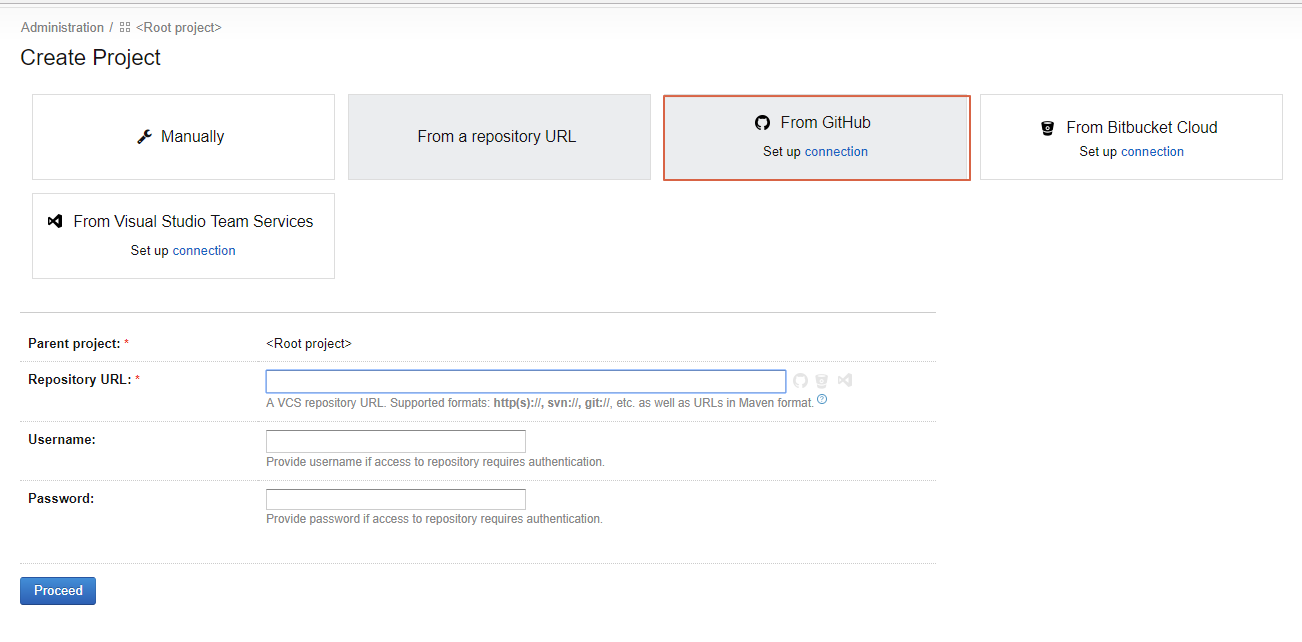
**Creating TeamCity Project**

Once the installation is done, TeamCity web user interface will be opened in the browser and we can create new TeamCity Project there. To do so, follow below steps:

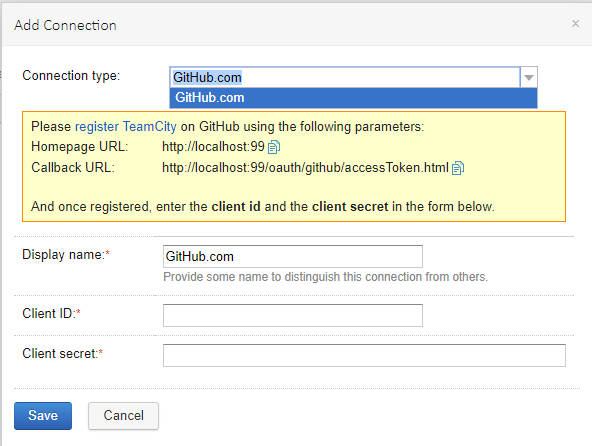
1. Once you have logged into TeamCity UI, Click on Create Project.



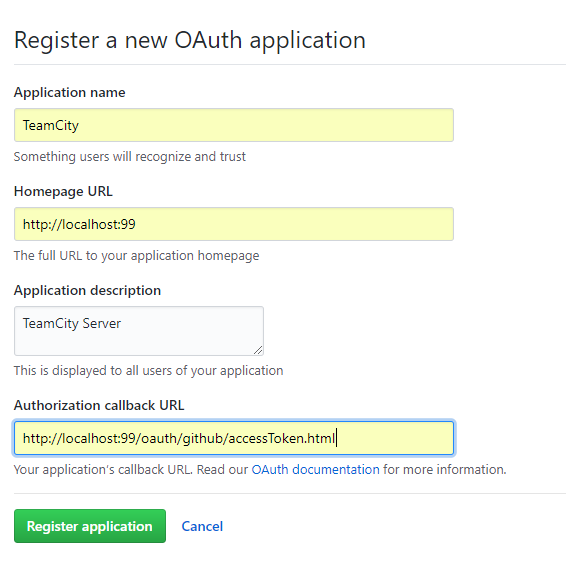
1. To connect our project from GitHub click on From GitHub on the next screen.



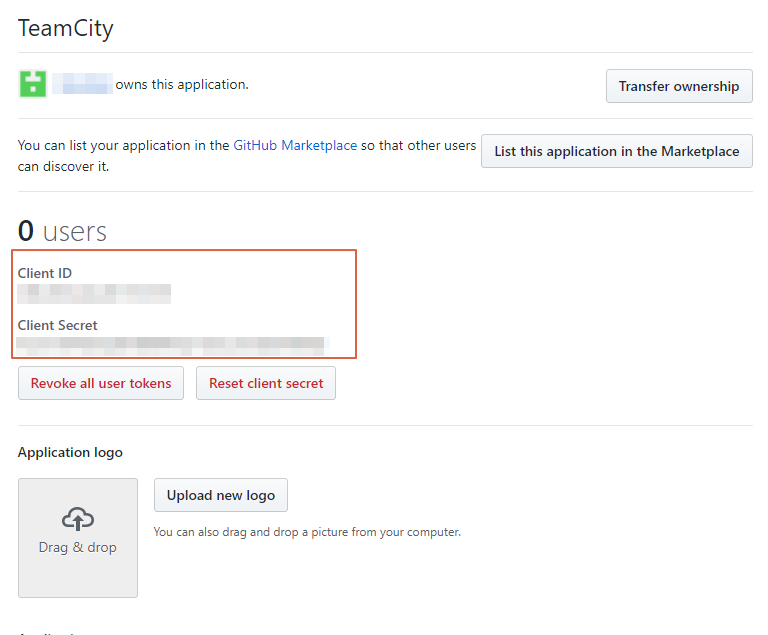
1. This will open a popup with some instructions to create add TeamCity application to your GitHub account.



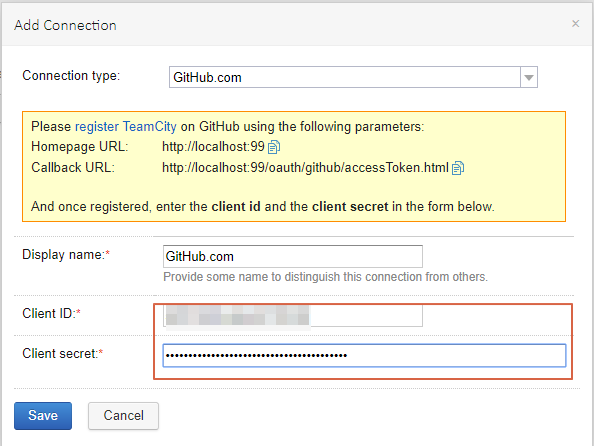
1. Click on register TeamCity link and it should take you to GitHub page where you can register a new OAuth app.
2. Give the details of the application and Homepage URl and Callback URL as shown in the image above (shown on the TeamCity screen) and register the OAuth app.



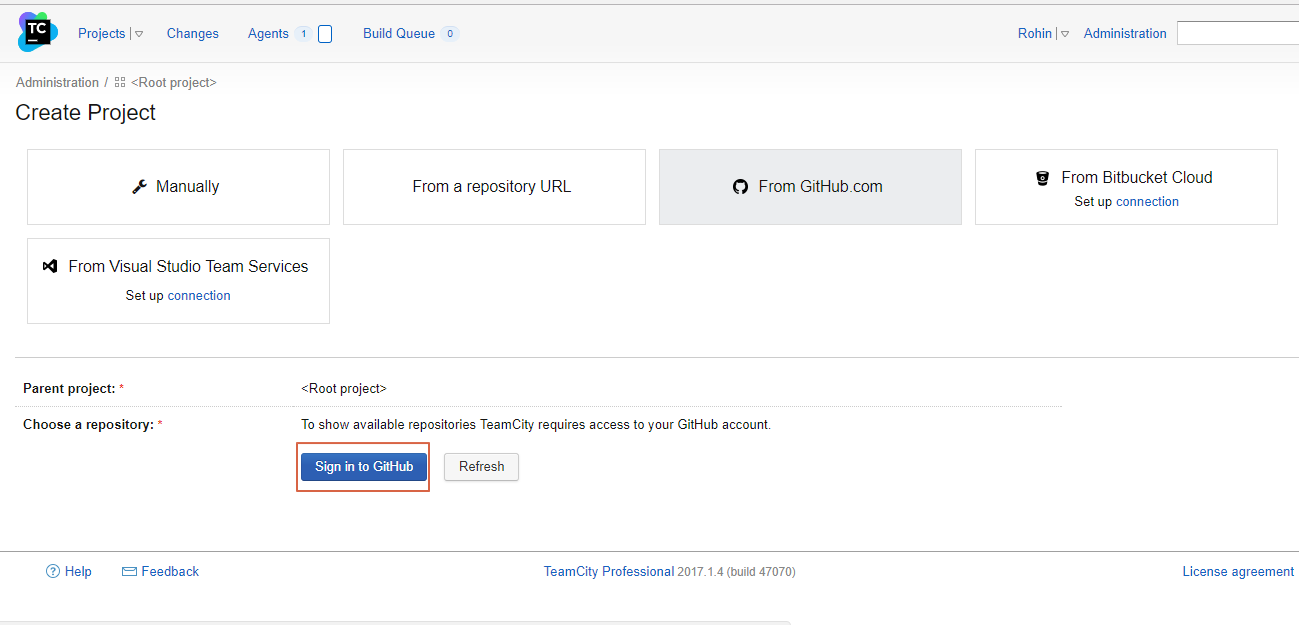
1. Once you register, on the next screen you’ll get Client ID and Client Secret, copy those details since it will be required on TeamCity project.



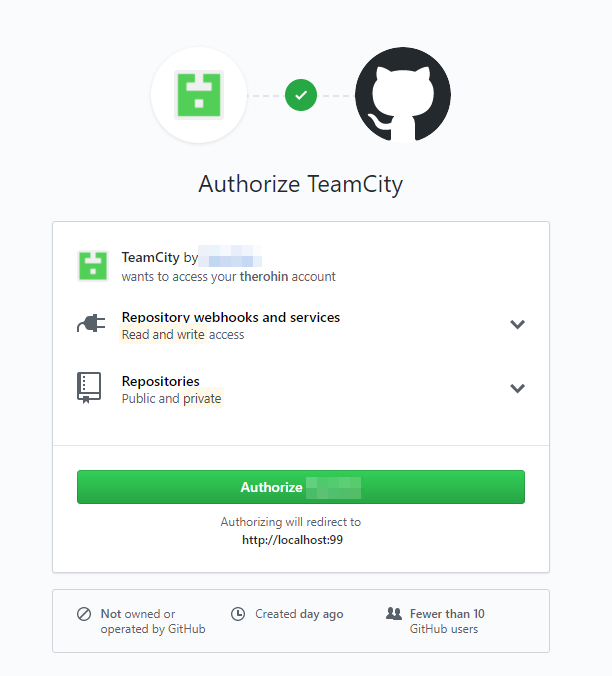
1. Come back to TeamCity and put Client ID and Client Secret in required fields and click Save.



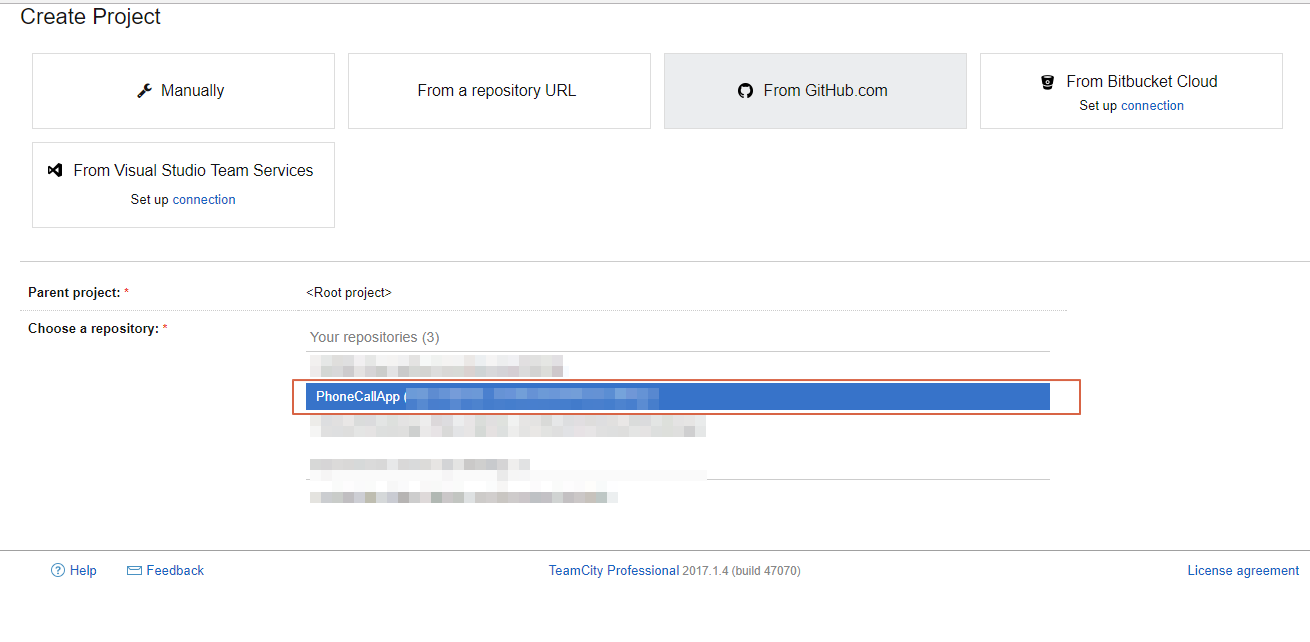
1. Next, you need to do a onetime sign in activity to allow TeamCity to use GitHub repositories. Click on Sign in to GitHub.



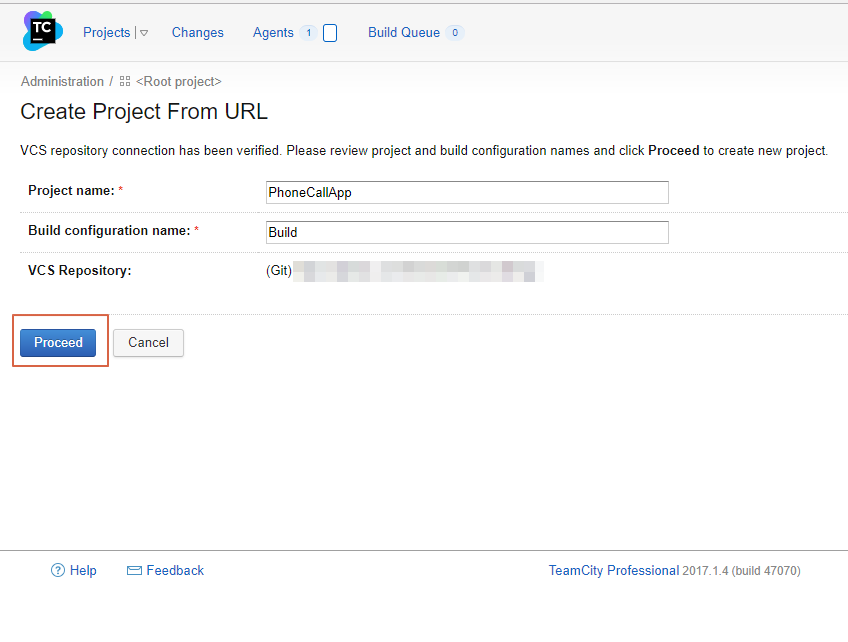
1. Authorize TeamCity App to use GitHub, click Authorize app.



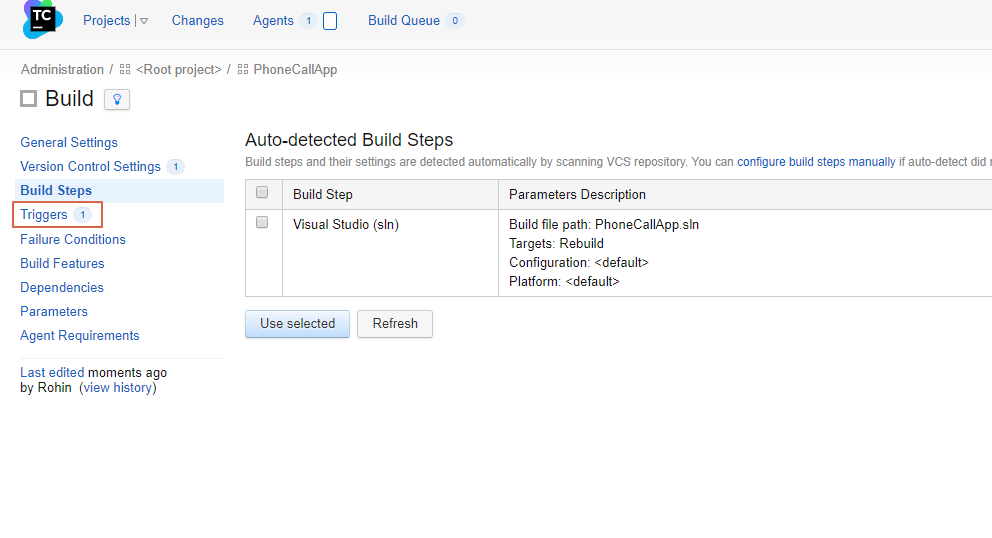
1. Once authorized, select the PhoneCallApp repository from the list of repositories shown on TeamCity.



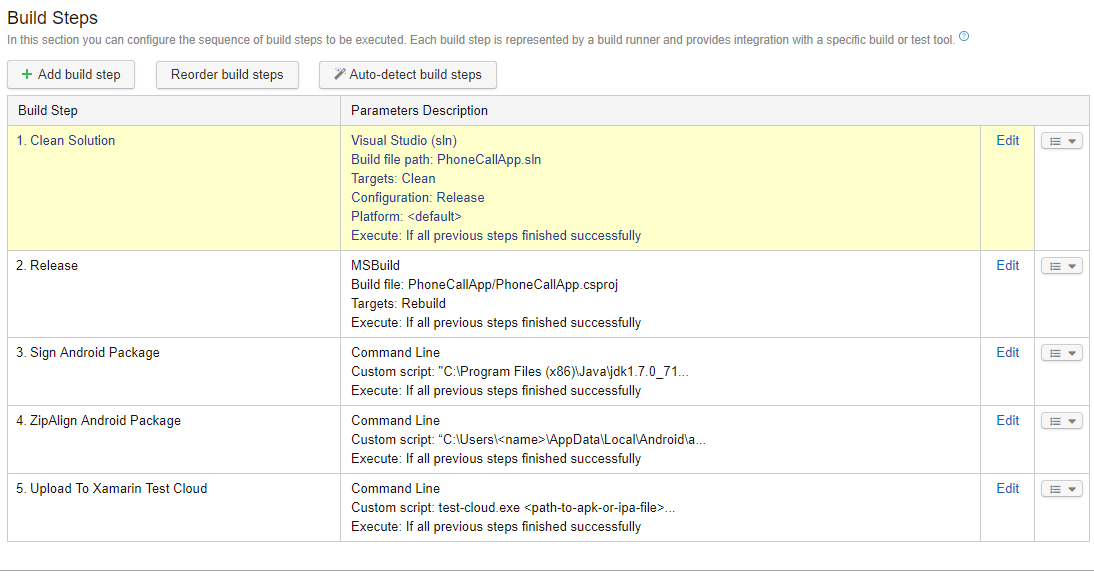
1. On the next screen, TeamCity would offer you to create a new project from the URL selected. Give it a name and click proceed.



1. This should create 2 things, 1st would be that it creates a trigger in TeamCity for each code check-in you do, it will trigger the build. Second would be that it creates build step from the repository automatically.



1. We would need to configure the build steps manually and use the build scripts described in previous section Create Build Script. Use those scripts described step by step in previous steps to create Build Steps in TeamCity.
2. Finally, your build steps should look like below image, comprising of all the steps mentioned earlier in Create Build Script section.



1. Now your TeamCity continuous build is ready, and a trigger is already configured for perform this build on each code check-in or whenever it finds any code changes in the repository. This finally provides you with an Android package that is ready to be distributed.

**Summary**

In this chapter, we learned about continuous integration and continuous delivery. We learned about various tools for continuous integration. We used TeamCity for implementing CICD with the Xamarin project developed in earlier chapters and learned how to create a project in TeamCity to automate the build process and finally get a distributable android package.

In the next chapter you’ll learn more about continuous distribution and delivery using Visual Studio Team Services.