Continuous Integration For .NET Projects With Jenkins

The automation toolchain of continuous integration begins with continuous integration in the version control system. Continuous Integration is a confusing topic to some, because, most believe that it has something to do with initiating the resource deployment once the team has built the project. But the story doesn't start there, the story starts at the moment someone hits "git push" -- or any similar alternate resource versioning tool (such as SVN).

The main primary focus of continuous integration and continuous deployment is to automate the process of software release. Removing the wait times needed by the teams for operations team wait time in order to maximize the output of the development teams. Sometimes this is termed under the umbrella of DevOps. DevOps is a huge and complex concept in itself, and most of the time DevOps is also misunderstood just like continuous integration. In any case, continuous integration is only a single step in the long-running DevOps toolchain, and in this post, you will learn what continuous integration happens to be, how to set up Jenkins environment for this process, and how to trigger it.

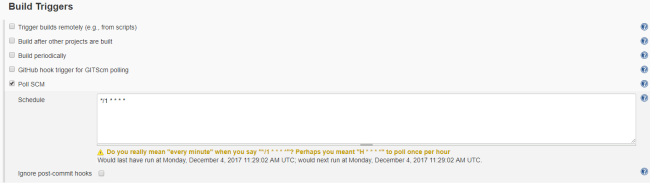
I will use Jenkins software to demonstrate the continuous integration workflow and how you can make the most out of it.

## **Jenkins Overview**

Although this post will not cover everything about Jenkins, it is needed to give you at least the overview of what Jenkins has for continuous integration processes.

Now, other platforms, such as Visual Studio Team Services, contain everything inside them by default. Which is why the system knows when something happens. Jenkins, on the other hand, is an extension or a solution for your processes where you control everything as a job. In Jenkins, you have to manually set triggers to build the project. Whereas, on other platforms, you can set the trigger, which is controlled internally and keeps a track of what has changed.

Jenkins on the other hands, provides a much-tailored solutions (but not as much as effective and efficient as a default event-driven one), in this format you what you do is you select an option from a really wide range of available options,



As seen in the image above, there are several options, such as,

* Timely build triggers
  + Useful in offices
  + Where you only build the project at the end of the day
* After other projects build
  + Can be used in environments, where your project is a dependent project, and you have to check whether you can utilize the latest build artifacts.
* Poll the source control
  + This is the one we are interested in;
  + You can set a timer, after which the Jenkins system will poll the version control to check if the repository has been updated or not.

This way, Jenkins provides you ability to run the build system and generate the artifacts for deployment. Jenkins is based on extensions, you install the extension for every kind of project. Jenkins itself is written in Java, thus for other project types, you would require consuming those extensions.

I will demonstrate usage of .NET build system in Jenkins workflow to build your .NET apps, as soon as there are changes in the repository.

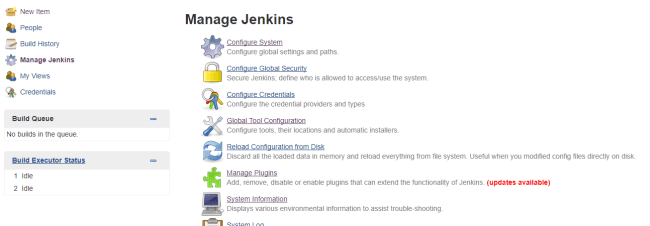
## **Setting up Jenkins**

This was the most painful step for me, and I am sure there are many others out there who have faced the same problems; such as msbuild.exe not found, or Git does not work and much more. In this section, you will learn how to manage the Jenkins.

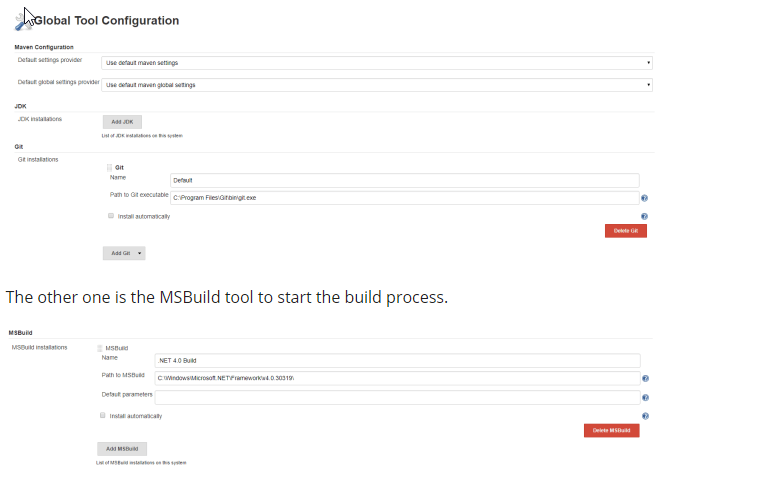
Setting up the Jenkins environment requires that you configure the plugins first, and then create a build system. I will assume that you already have everything set up and ready to be used.

* Git version control must be installed
* .NET framework and the MSBuild app should be there
* MSBuild extension in Jenkins must be installed
* Anything else; such as the MSTest, which you can then use later on by yourself.

You should go to the Global Tool Configurations for the plugins, and configure the default values for the system.



The following configurations must be done before you create the build system and set up everything (you can have an existing build configuration too if you already have created it).



In case you do not see this MSBuild, make sure that you followed the steps and installed everything needed for this. Such as, the MSBuild plugin from the Jenkins plugin library,



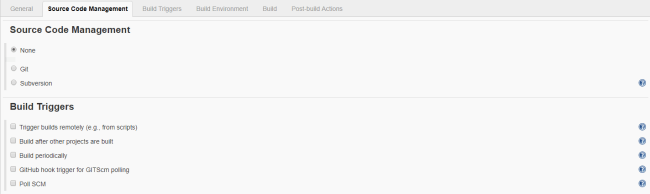
Once everything was done, you can continue to the next section and create the continuous integration staging process.

## **Continuous Integration With Jenkins Setup**

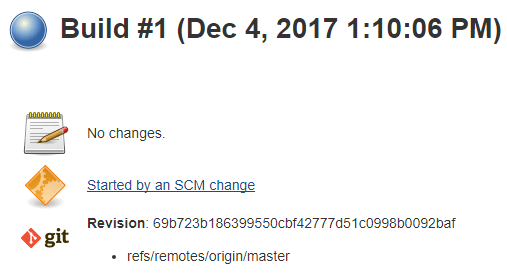
We have set up our system by now, and need to create a new build definition. The build definitions are the overall process that controls how your project is built. Each build definition has several jobs that run and process your overall workflow. In our build system, we need to configure a few things,

* The repository to listen to for any changes
* Setup of the MSBuild and the solution.

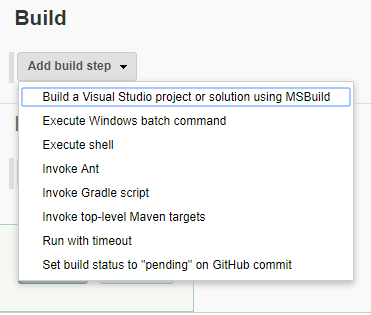
In these two steps, you will set up the repositories and everything other than that.



Configuring these settings will let you control the continuous integration workflow in Jenkins. Just set up your own configurations for Git and the settings, which will let you manage how your project is built. Without a build step, your build will always pass without any error and any action performed. Such as the following one, although you can see that it does pull something off the git version control.



Lastly, just set up a build step and finalize the settings. In this step you will need to add the path of the solution (.sln) file, that you have for your project. This will create a step.



You can have as many steps in your build definition as much as you want. One more thing, if you cannot see this step then you need to install the plugin. Also, if you install more plugins, you can see other build steps here as well, such as for other project types.

## **Triggering the Build**

Now that our basic functionality has been setup, we can go ahead and make some changes in the repository and then push the changes for Jenkins to capture and trigger the build system. I would just add a comment that will allow me to commit the changes, and trigger the build.



After I had done the git commit, I was able to get the code built by the Jenkins system. The first error was that the solution file and the csproj file were in use by Visual Studio; after closing it, I was able to have the project built and everything ready for me.

## **Onwards**

Continuous Integration is merely the first step toward the overall deployment and release management lifecycle. I did not cover the testing scenario to test the build artifacts of your project. You can modulate all of that and try it yourself.

The next steps are for the deployment. In Jenkins, you can archive the project's bin content and store it somewhere, or upload via FTP to a remote directory from where your customers can download and update their software.

## **Continuous** Deployment with Jenkins & .NET

As a .NET developer you are probably using Visual Studio. Hitting F5 in Visual Studio compiles the source code and starts the debugger. If you are like me you will hit F5 many times during development and bug hunting.

But what happens when the code is ready for prime time? Deployment happens!

When I needed to publish an application I often choose the laziest option: manually copying the executable to the destination, manually installing and manually starting the application.

It’s “fun” the first few times, but in the end it gets really difficult; particularly keeping track of changes and manual updates/deployments. To be honest, this manual approach does not scale very well: it takes unnecessary time and it’s prone to human errors in the deployment process.

A more mature option would be to have an automated infrastructure to handle deployment. Such infrastructure is often referred to as Continuous Integration/ Continuous Deployment.

Using Continuous Integration/Continuous Deployment, enables a workflow like this:

* Code, build and test locally on your dev. box (as usual)
* Check source code in to a development branch in your favourite Source Control Management System.
* Optional code review by team members.
* Merge your changes to the main (release) branch.
* The CI Server will detect the changes to the main branch and download, compile and deploy/install your application on the release server.

There are no manual steps: it’s all scripted and automatic.