

Continuous Integration (CI) for .NET applications using Jenkins

Having a Continuous Integration (CI) and Continuous Delivery (CD) process during the various stages of development helps to reduce risks, catch bugs more quickly and rapidly deploy software to various environments. These benefits result in developers being able to implement business requirements and user needs more quickly and turning the process of releasing software into a business advantage.

Tools

There are many tools out there to accommodate CI and CD for your application such as TeamCity, Jenkins, GitLab CI, Bamboo, Team Foundation Server, etc. In this case we're going to setup CI (and later on CD) using [Jenkins](#) and [Atlassian Bitbucket](#).

Jenkins is an upcoming popular open source Continuous Integration and Continuous Delivery tool that provides you to automate builds and deployments for various programming languages. There is an extensive [plugin library](#) available that allows you to use and configure tools like MSBuild and MSTest in your job with ease.

Automate IT

To actually create and setup a job, a few prerequisites are required:

Prerequisites

- Setup a VM or physical server running Windows.
- Install [Jenkins](#).
- Install [Visual Studio](#) or [Microsoft Build Tools](#).
- Install [Git for Windows](#) (or any other code repository client tools).
- Create a [Bitbucket](#) (or any other code repository) account and create a repository for your code.
- Install the [MSBuild plugin](#) for Jenkins.
- Install the [MSTest plugin](#) for Jenkins.
- Install the [Git plugin](#) for Jenkins.

To compile and test your .NET application using Jenkins, [MSBuild](#) and [MSTest](#) are obviously required. I recommend installing Visual Studio which contains all the necessary tools to compile, test and measure your application rather than installing the trimmed down Microsoft Build Tools, especially if you are using the Microsoft Fakes framework in your unit tests. Installing Visual Studio Ultimate or higher next to Jenkins would be the best approach to cover that scenario so you are able to run those unit tests using Jenkins.

Configuring plugins

The installed plugins must be configured in order to use them. Go to the **Configure System** section in Jenkins and fill in the paths to the executables for the Git, MSBuild and MSTest plugins.

Step 1: Create a new job

To create a new job in Jenkins, navigate to the Jenkins web interface using your favorite browser and click on the **New item** button. This navigates to the screen below:

Item name

- Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- MultiJob Project**
MultiJob Project, suitable for running other jobs
- Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- External Job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).
- Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Copy existing Item**
Copy from

Fill in your project name, select **Freestyle project** and click **OK**. The Freestyle project job type consists of all the necessary elements such as SCM integration, build scripts and optional steps required for our project.

Step 2: Connect your repository

The next step is to connect your repository so Jenkins knows where to find and pull your code. Scroll down to the section **Source Code Management** and select **Git**:

Source Code Management

None
 CVS
 CVS Projectset
 Git

Repositories

Repository URL ?

Please enter Git repository.

Credentials ?

Branches to build

Branch Specifier (blank for 'any') ?

Repository browser ?

Additional Behaviours

Subversion
 Team Foundation Server

Fill in your

repository URL and create a user within Jenkins that has at least read access to the configured repository. In this case we will be pulling and monitoring the `master` branch for changes.

Now, Jenkins is not pulling code automagical, first a build trigger must be created. Bitbucket offers integration with Jenkins via [webhooks](#) or [repository hooks](#).

A `hook` can be made to trigger a job when SCM changes occur. I would recommend configuring a hook over polling manually to reduce the number of jobs running on Jenkins. If you want to manually poll your repository, just check the `Poll SCM` or `Build periodically` checkbox in the `Build Triggers` section.

Build Triggers

- Trigger builds remotely (e.g., from scripts)
- Build after other projects are built
- Build periodically
- Build when a change is pushed to BitBucket
- Build when another project is promoted
- Poll SCM

Step 3: Add a build step

To actually compile your project or solution that Jenkins pulled from your repository, a build step must be created. Scroll down to the `Build` section, select `Add build step` and choose `Build a Visual Studio project or solution using MSBuild`. The following step is created:

Build

Build a Visual Studio project or solution using MSBuild

MSBuild Version	(Default) ▾	
MSBuild Build File	YourProject.sln	?
Command Line Arguments	<div style="border: 1px solid #ccc; height: 40px;"></div>	?
Pass build variables as properties	<input type="checkbox"/>	?
Continue Job on build Failure	<input type="checkbox"/>	?
If warnings set the build to Unstable	<input type="checkbox"/>	?

[Delete](#)

Add build step ▾

Select

the version of MSBuild you want to use, in this case we will use the `Default` one. And enter the path to the solution or project file that has to be compiled using MSBuild. In our case we are compiling `YourProject.sln`. The solution file is in the root of the working directory, so no additional path prefixing is required.

Step 4: Run unit tests

The last thing to do is to create a build step to run your unit tests after your code has been successfully compiled. To do this, select `Add build step` again and choose `Run unit tests with MSTest`:

Run unit tests with MSTest

MsTest Version	(Default) ▾	
Test Files	%WORKSPACE%\TestsProject\bin\Debug\TestProject.dll	?
Test Categories		?
Result File Name		?
Command Line Arguments		?
Continue on failed tests	<input type="checkbox"/>	?

[Delete](#)

Add build step ▾

Select the version of MSTest you want to use, again in our case we will use the `Default` one. And enter the paths to the dlls containing your unit tests.

Next step

And there you have it, a fully automated Jenkins job that pulls code from the repository when a SCM change occurs, compiles it and runs unit tests. Ofcourse these are the minimal steps required to create a Continuous Integration job for your project, many other options can be configured such as pushing notifications to Slack, parameterizing builds, creating reports, etc.

The next step is to configure Continuous Delivery (CD) using Jenkins,

Continuous Delivery (CD) for .NET applications using Jenkins

The key to delivering software more rapidly and more frequently is having a continuous delivery process which makes deploying your application as easy as pushing a button. When it's easy to deploy your application, bugfixes and new features end up in production way faster and in a more agile way than the old fashion way of manually creating a package and handing it over to someone from the ops team who would then deploy it to the requested environment.

Delivering software the old fashion way is often prone to human error and takes a lot more time forcing developers to create bigger releases with a larger number of features and bugfixes which in turn also takes more time. This is obviously bad for business because end-users will have to wait much longer to be able to enjoy new releases, and in the meantime they could be getting frustrated by potential bugs which aren't going to be resolved very quickly.

One-click deploy

The main goal is to be able to deploy our application by the click of a button. To do that, we're going to be using [Jenkins](#) and add a deployment process to our [previous](#) CI job to deploy our application to [Microsoft Azure](#).

Prerequisites

- Follow my [previous](#) blogpost to create a continuous integration job using Jenkins.
- Create a [Microsoft Azure Account](#).
- Create a new [Azure App Service](#) in the Azure portal.
- Create a shared network location or setup an [Artifact Repository](#) to store the build artifact.
- Install the [Promoted Builds](#) plugin for Jenkins.

We're going to use the Promoted Builds plugin to create a build artifact of our application and deploy it to Azure via [Web Deploy](#). Because the Azure App Service we've created already supports Web Deploy, there is no need to install it. In this case Azure is our production environment. I recommend also having a development, test and acceptance environment to be able to test and preview your application before it gets deployed to production.

Artifact repositories

Because we're creating a build artifact, it is wise to store that artifact in a binary repository or on a shared location. This way you're not only able to keep track of all your releases, but also to rollback a previous version. There are a few good repositories out there such as [Archiva](#), [Artifactory](#), [Nexus](#) and [Package Drone](#). [Here](#) is a good comparison chart for you to decide which repository suits you best.

Step 1: Create a promotion process

Navigate using your favorite browser to the Jenkins job you've created and tick the **Promote builds when...** checkbox. The following section appears:

Promote builds when... ?

Promotion process

Name

Required

Icon

Restrict where this promotion process can be run

Criteria

- Only when manually approved ?
- Promote immediately once the build is complete ?
- Promote immediately once the build is complete based on build parameters ?
- When the following downstream projects build successfully ?
- When the following upstream promotions are promoted ?

Actions

In

this case we're going to create the process of deploying our application to a production environment so the name of this process is going to be **Production**. Tick the **Only when manually approved** checkbox because we don't want to trigger the process automatically.

Typically you can tick the **Promote immediately once the build is complete** checkbox for a development promotion process. This means that when a build is succeeded and all unit tests pass, a build artifact is created immediately and is then deployed to the development environment.

If you have multiple promotion processes, i recommend ticking the **When the following upstream promotions are promoted** checkbox and fill in the previous promotion process name (e.g. **Acceptance**) that has to be successfully executed before being able to approve the next promotion process. This way you will force your application being deployed to other environments first such as development, test and acceptance before reaching production. An example would be **Development > Test > Acceptance > Production**.

Step 2: Create a build artifact

To create a build artifact click the **Add action** button in the **Actions** region of your promotion process and select **Build a Visual Studio project or solution using MSBuild**:

The screenshot shows the configuration for the 'Build a Visual Studio project or solution using MSBuild' action. The 'MSBuild Version' is set to '(Default)'. The 'MSBuild Build File' is 'YourProject.csproj'. The 'Command Line Arguments' field contains the following text: `/T:Clean;Build;Package /p:Configuration=Release /p:OutputPath="obj\Release" /p:PrecompileBeforePublish=true`. There is a checkbox for 'Pass build variables as properties' which is currently unchecked. At the bottom, there are buttons for 'Add action', 'Advanced...', 'Delete', and 'Add another promotion process'.

This step is almost equal to our build step which we created in our CI process, but now we're going to create a web deploy package in stead of compiling the solution. To create that package, we have to add a few command line arguments:

- `/T:Clean;Build;Package` to create a Web Deploy package.
- `/p:Configuration=Release` to use the release configuration.
- `/p:OutputPath="obj\Release"` to put the artifact in the obj\Release folder of the current project.
- `/p:PrecompileBeforePublish=true` to precompile our application.

Step 3: Store the build artifact

Next we have to store our build artifact we've just created. In this case we will keep it simple and just copy it to a shared network location. Click the **Add action** button again and select **Execute Windows batch command**:

The screenshot shows the configuration for the 'Execute Windows batch command' action. The 'Command' field contains the following text: `xcopy YourProject\obj\Release_PublishedWebsites\YourProject_Package\YourProject.zip "\\networklocation\Production\#%PROMOTED_ID%" /s /e /y`. Below the command field, there is a link that says 'See the list of available environment variables'. At the bottom, there are buttons for 'Add action', 'Delete', and 'Add another promotion process'.

Because we're just copying the artifact to another location, i've created a `xcopy` command that copies the build artifact to a specified network location and create a folder with the Jenkins build number.

Step 4: Deploy the build artifact

To be able to deploy to Azure via Web Deploy you first have to download the `Publish Profile` of your Azure App Service. Go to the Azure Portal, navigate to your App Service and click the `Get Publish Profile` button. You should get something like this:

```
<publishData>

  <publishProfile profileName="YourProject - Web Deploy"

    publishMethod="MSDeploy"

    publishUrl="yourproject.scm.azurewebsites.net:443"

    msdeploySite="YourProject"

    userName="$YourProject"

    userPWD="someRandomPassword"

    destinationAppUrl="http://yourproject.azurewebsites.net"

    SQLServerDBConnectionString=""

    mySQLDBConnectionString=""

    hostingProviderForumLink=""

    controlPanelLink="http://windows.azure.com"

    webSystem="WebSites">

  <databases />

</publishProfile>

<publishProfile

  profileName="YourProject - FTP"
```

```
publishMethod="FTP"

publishUrl="ftp://waws-prod-sn1-
027.ftp.azurewebsites.windows.net/site/wwwroot"

ftpPassiveMode="True"

userName="YourProject\YourProject"

userPWD="someRandomPassword"

destinationAppUrl="http://yourproject.azurewebsites.net"

SQLServerDBConnectionString=""

mySQLDBConnectionString=""

hostingProviderForumLink=""

controlPanelLink="http://windows.azure.com"

webSystem="WebSites">

<databases />

</publishProfile>

</publishData>
```

XML

We are going to use the `Web Deploy` profile in our next action. Because there is no `MSDeploy` plugin for Jenkins click `Add action` and select `Execute Windows batch command`:



This is a pretty large command using the MSDeploy executable, so let's go over the command line arguments:

- `-verb:sync` to update our web application.
- `-`
`source:package="YourProject\obj\Release_PublishedWebsites\YourProject_Packa`
`ge\YourProject.zip"` to use the build artifact for deployment.
- `-`
`dest:auto,computerName='https://yourproject.scm.azurewebsites.net:443/msdeploy.`
`axd?site=yourproject',UserName='$YourProject',Password='someRandomPassword`
`',AuthType='Basic',includeAcls='false'` to set the destination for our deployment using the data from the downloaded Publish Profile. `includeAcls` is set to false because we don't want to deploy file ownership information.
- `-setParam:name='IIS Web Application Name',value='yourproject'` to override the web application name with the one specified in the Azure App Service.
- `-disableLink:AppPoolExtension` to disable setting the AppPool because it is already set by Azure.
- `-disableLink:ContentExtension` to disable deploying the contents of optionally configured virtual directories.
- `-disableLink:CertificateExtension` to disable deploying optionally included certificates.

That's it for the production process. Once a build is completed and successfully passed the configured unit tests, the only thing you have to do to deploy your application is go to your build in Jenkins, click the `Promotion Status` button in the left option pane and click the `Approve` button of the promotion process you want to execute.

Rollback

But what about rollback? A similar promotion process can be created. Instead of creating and copying a build artifact, we are only going to deploy an archived artifact.

Click the button **Add another promotion process** and call it **Rollback production**. Again tick the **Only when manually approved** checkbox. But now we are going to add a parameter to our process by clicking the button **Add Parameter** and selecting **String Parameter**:

The screenshot shows the configuration for a Jenkins promotion process named "Rollback Production". The "Name" field is "Rollback Production" and the "Icon" is "Gold empty star". There is a checkbox for "Restrict where this promotion process can be run" which is unchecked. Under the "Criteria" section, the checkbox "Only when manually approved" is checked. Below it is an "Approvers" field. Under "Approval Parameters", a "String Parameter" is added with the name "BuildVersion" and a description "Enter the build number to restore". There is a "Delete" button for this parameter. Below the parameter list is an "Add Parameter" dropdown menu. At the bottom of the criteria section are four unchecked checkboxes: "Promote immediately once the build is complete", "Promote immediately once the build is complete based on build parameters", "When the following downstream projects build successfully", and "When the following upstream promotions are promoted". The "Actions" section has an "Add action" dropdown menu. At the very bottom are two buttons: "Add another promotion process" and "Delete this promotion process".

Because we want to rollback a specific release, we're going to use the Jenkins build number to select an artifact from the shared network location. Again click **Add action** and select **Execute Windows batch command**:

The screenshot shows the configuration for the "Execute Windows batch command" action. The "Command" field contains the following text: `"C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe" -verb:sync -source:package="\networklocation\Production\%BuildVersion%\YourProject.zip" -dest:auto,computerName='https://yourproject.scm.azurewebsites.net:443/msdeploy.axd?site=yourproject',UserName='$YourProject',Password='someRandomPassword',AuthType='Basic',includeAcls='false' -setParam:name='IIS Web Application Name',value='yourproject' -disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension`. There is a "Delete" button for this action. At the bottom left is an "Add action" dropdown menu. At the bottom right are two buttons: "Add another promotion process" and "Delete this promotion process".

Copy the MSDeploy command from Step 4 and change the **-source** parameter to point to the artifact from the shared network location in stead of the one created in the obj folder: -

```
source:package="\networklocation\Production\#%BuildVersion%\YourProject.zip".
```

As you can see, the `BuildVersion` parameter is used in the path to point to the artifact created from that specific build. You can pass the parameter when approving your promotion process at the Promotion Status screen.

Thoughts

Instead of deploying to Azure, you could also deploy to a local server or even to a docker or windows container using an orchestration tool such as [Microsoft Azure Service Fabric](#) or [Pivotal Cloud Foundry](#).

As i mentioned earlier, create several promotion processes for your environments. Be sure to archive artifacts if needed, as archiving development releases is not really necessary. Make promotion processes dependent on each other to ensure your application is always being deployed though your entire DTAP street before reaching production.