**Continuous Integration & Delivery For GitHub With TeamCity**

This is an end to end tutorial for setting up CI and CD for a GitHub project using TeamCity.This is the technology stack I am using in the project, [Humanizer](https://github.com/MehdiK/Humanizer), for which I am setting up the TeamCity project:

* .Net: the programming language used for Humanizer.
* Git and GitHub: Humanizer repository is git and the project is [hosted on GitHub](https://github.com/MehdiK/Humanizer).
* XUnit: the testing framework I am using in Humanizer.
* NuGet: the package manager for .Net which is where [Humanizer is deployed to](http://nuget.org/packages/humanizer).

That said many of the topics in this post are more or less applicable to other technologies.

**What this post is about**:

* [Creating a TeamCity project](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#creating-teamcity-project)
* [Setting up Continuous Integration](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#setting-up-ci)
  + [Getting the source code](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#getting-source-code)
  + [Building the solution](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#building-the-solution)
  + [Running the tests](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#running-the-tests)
  + [Creating a NuGet package](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#creating-nuget-package)
  + [Rewriting assembly versions](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#rewriting-assembly-version)
  + [Setting up build triggers](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#ci-build-trigger)
  + [Setting up build notification for GitHub pull requests](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#pr-notification)
  + [Showing build status icon](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#build-status-icon)
  + [Wrapping up Continuous Integration](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#ci-wrap-up)
* [Setting up Continuous Delivery](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#setting-up-cd)
  + [Source Control Settings](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#cd-source-control-settings)
  + [Publishing NuGet packages](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#publishing-nuget) (to [nuget.org](http://nuget.org/) and [symbolsource.org](http://symbolsource.org/))
  + [Build triggers](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#cd-build-triggers)
  + [Getting deployable artifacts](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#getting-deployable-artifacts)
  + [Wrapping up Continuous Delivery](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/#cd-wrap-up)

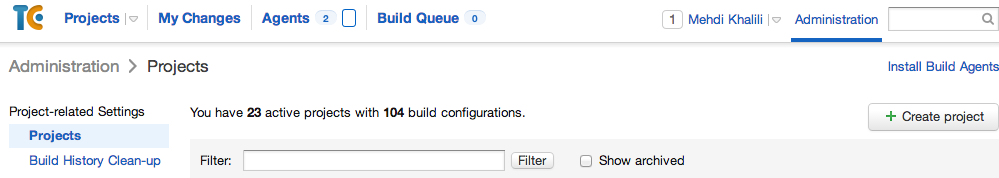
**What this post is not about**:

* Installing TeamCity on your servers
* Setting up build agents, users and roles.

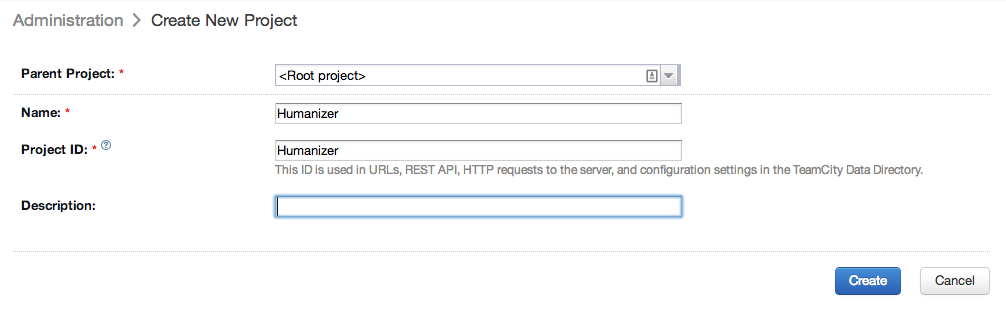
For this post, I assume that you have a running TeamCity server and a user with System administrator rights and you’re logged into the admin console.

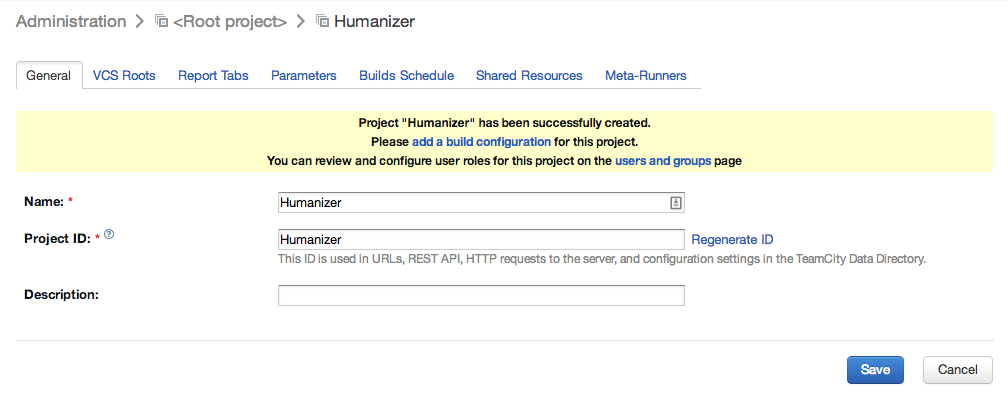
**Creating a TeamCity project**

So before anything else we need to create a TeamCity project which is a simple grouping of build configurations. On the TeamCity admin console go to Administration (and click on the Projects from the left navigation bar). That takes you to a page showing a list of projects. There is also a ‘Create Project’ button (if you are System Administrator) to create new projects:



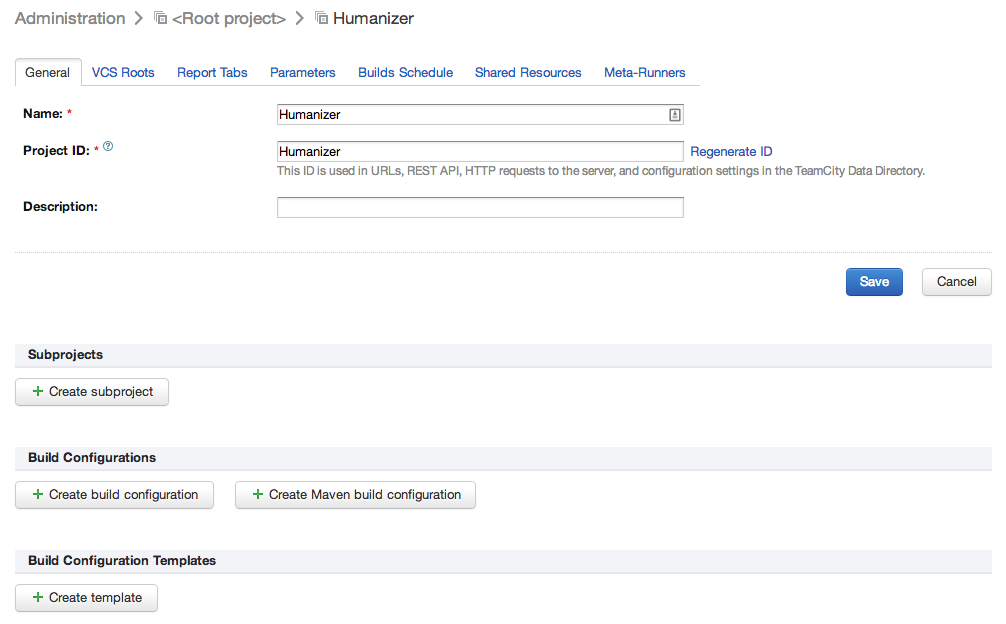
After clicking on the ‘Create Project’ button you see the following page where you can enter your project details:



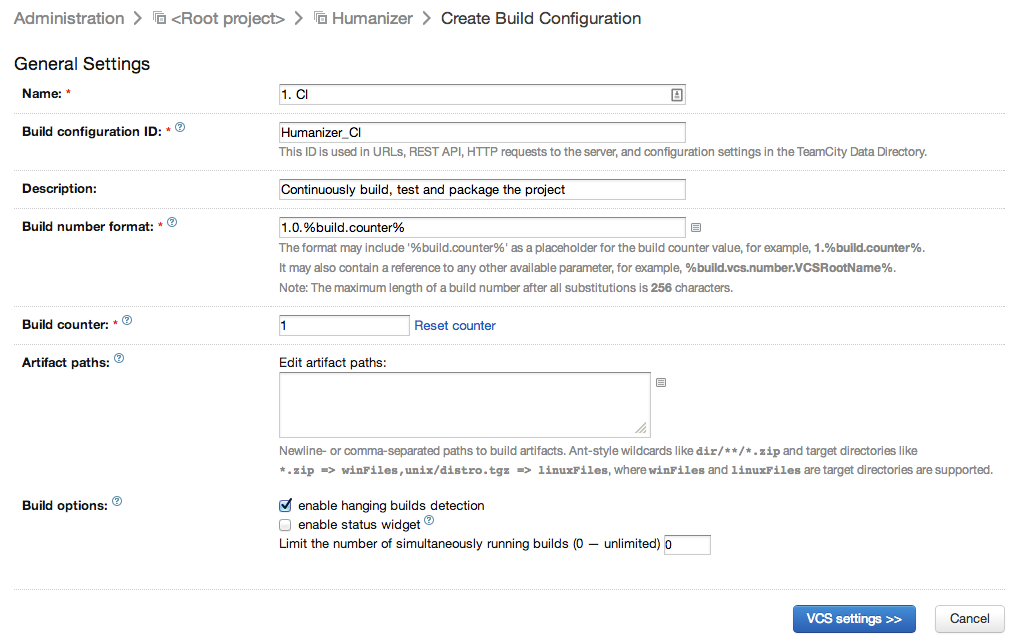
After you create the project, you are taken to the project home page where you can setup the build configurations

**Setting up Continuous Integration**

So you now have a TeamCity project and want to setup [Continuous Integration](http://www.martinfowler.com/articles/continuousIntegration.html) for your project: Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. The first step is to create a build configuration for Continuous Integration. You can do so by clicking on the ‘Create build configuration’ button on the project home page:



After clicking on the button you will be taken to the ‘Create build configuration’ page:

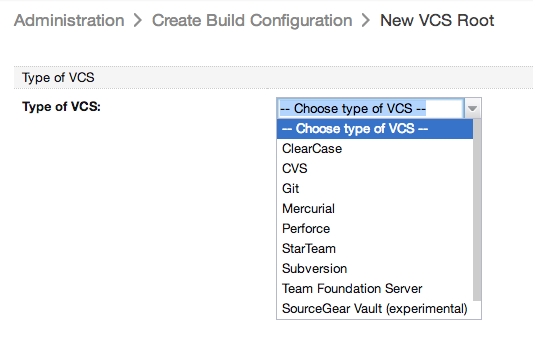


**The settings:**

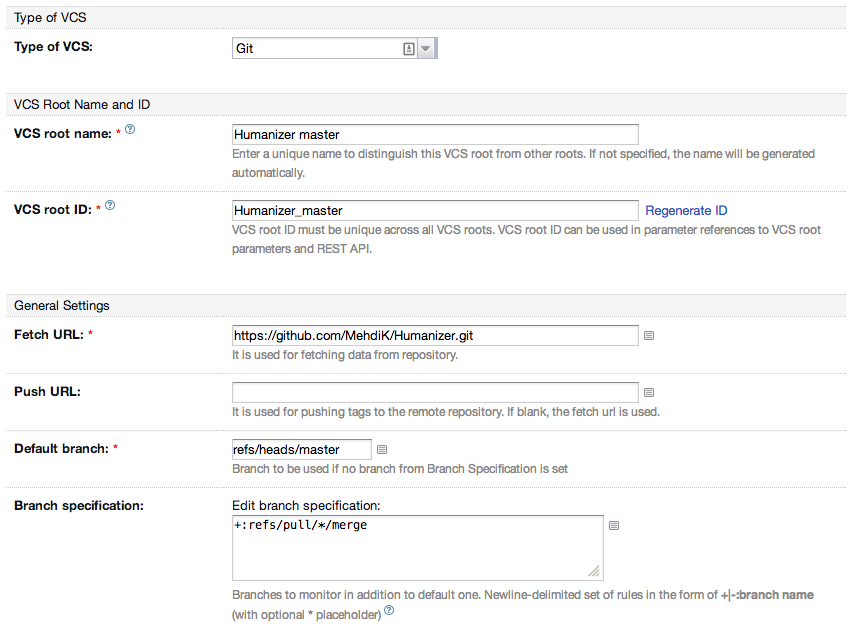
* I name the build setting ‘1. CI’ so the next person knows that this is a CI build configuration and also that it’s the first build config.
* ‘Build configuration ID’ is a unique identifier for this build configuration. We will see where this is useful later.
* ‘Description’ is optional; but having a description there makes it easier for the person maintaining the project.
* ‘Build number format’ is the build number. You can hardcode this value or could use the TeamCity provided value %build.counter% as part of your build number. I use 1.0.%build.counter%because the project I am setting up, [Humanizer](https://github.com/MehdiK/Humanizer), is currently on version 1 and I want the future builds to continue from there. This is one setting that you will be changing rather often: in my case any time I want to release a new version with a different major or minor.
* ‘Build counter’ starts from 1 and increases on each build. This is basically the value that gets injected into %build.counter%.

**Source control settings**

Once your build config is created you should set the ‘VCS settings’ so TeamCity knows how to get the code needed for the build. Different VCS engines have different settings so the first step is to pick the VCS you want to use - in our case Git (for GitHub):

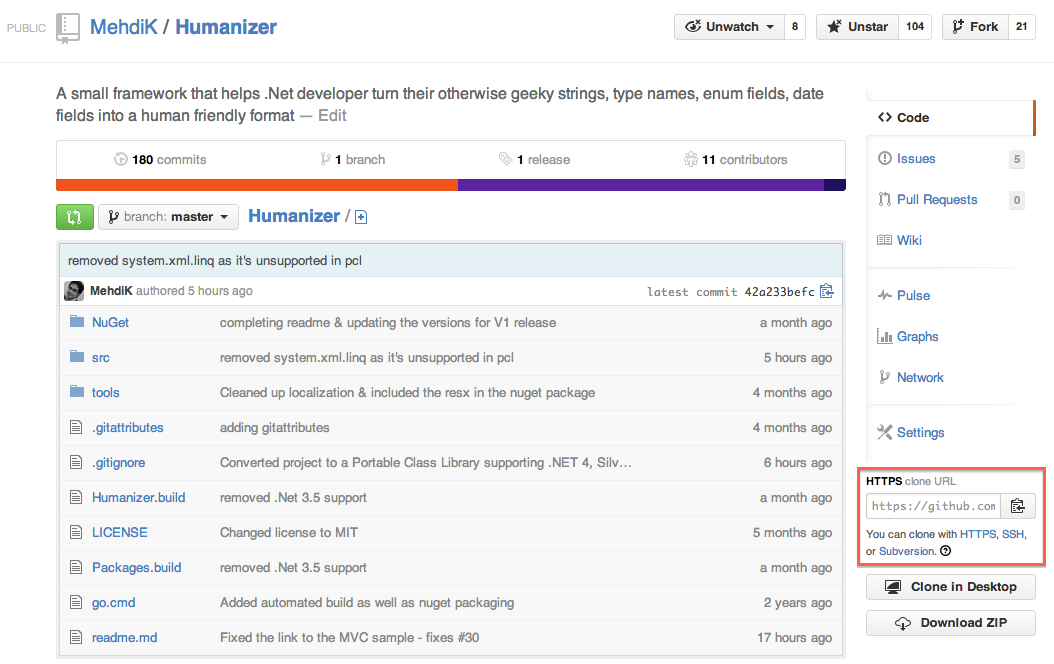


Once you choose Git you are provided with a page to setup your git repository connection:



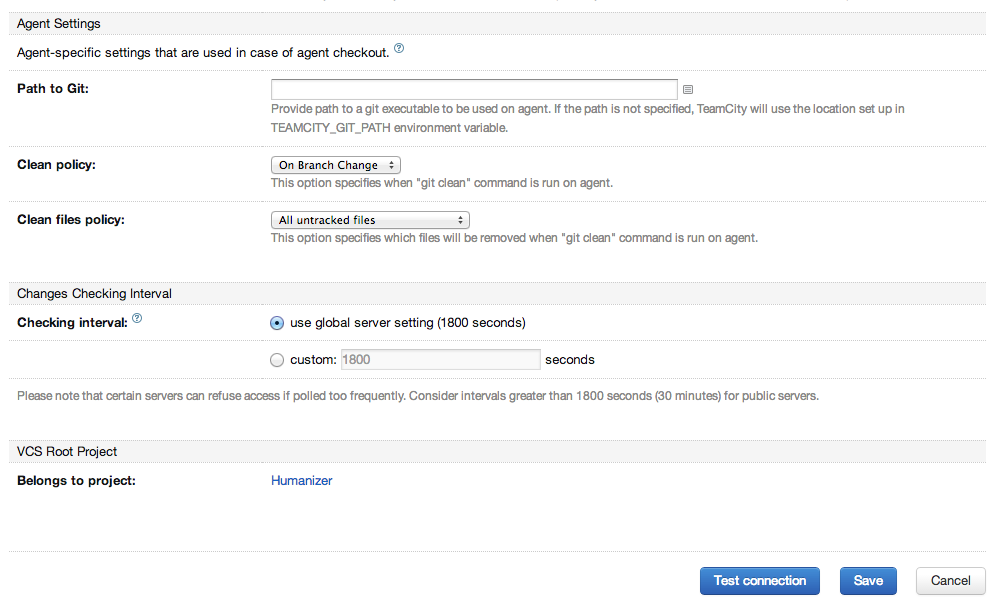
The important [git VCS settings](http://confluence.jetbrains.com/display/TCD8/Git+%28JetBrains%29) are:

* VCS root name and ID: use a unique name preferably related to your project so it’s easy to spot it amongst other VCS roots you might have in your TeamCity.
* Fetch URL: this tells TeamCity where it should look for the source code. If you’re using GitHub you can grab this from your project’s GitHub home page (the SSH one requires authentication; so you might grab the https URL):



* ‘Default branch’ should be set to your master branch which is where your CI should focus more. So we set it to refs/heads/master which is the git master branch. If you’re wondering where that comes from run git show-ref master command on your git repo.
* We want to also monitor pull request branches so we set the ‘Branch specification’ to+refs/pull/\*/merge which uses wildcard to monitor pull request merged branches.

I leave the rest of the settings to their default values. Before you save, make sure to ‘Test connection’ so that TeamCity can find your repository with the settings you have provided. The button is located down the bottom of the page:



Once you create a VCS root you can reuse it across build configurations.

**Create build steps**

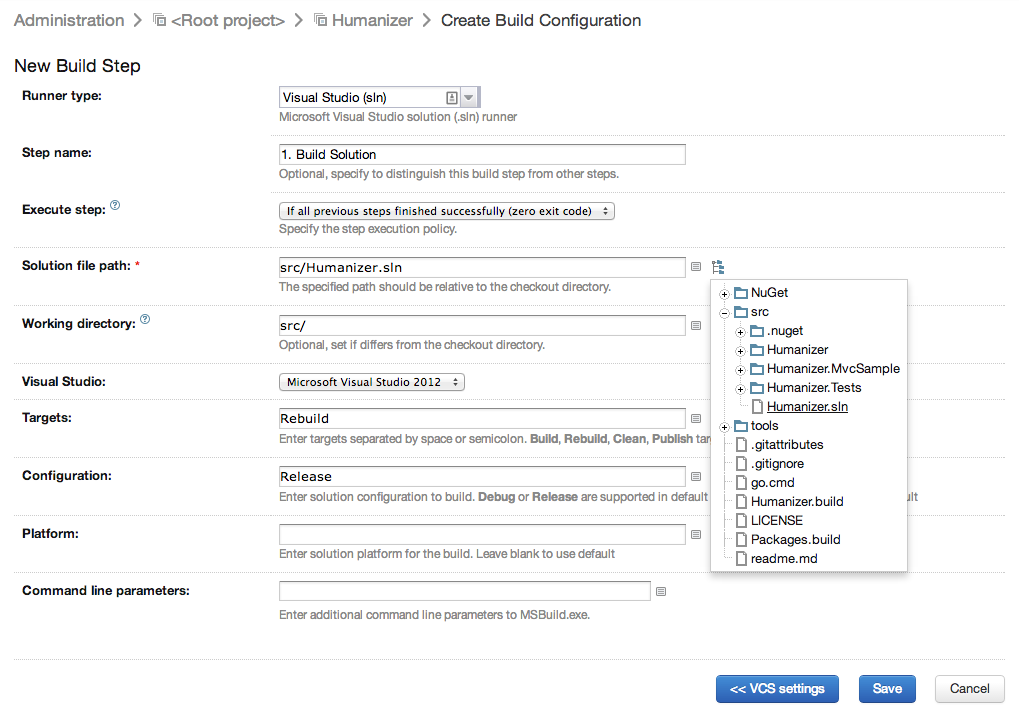
You have a build configuration attached to a source control. We can now create the build steps. I am going to create three steps:

1. Build Solution: to build the code
2. Run Tests: to run the tests
3. Pack NuGet: to create NuGet packages from the artifacts generated in the previous steps

To create a build step you should click on the ‘Add build step’ button. Much like the VCS setting, you should first specify what sort of step you are creating before you’re provided with the specific settings for that action.

1. Build Solution

To build a .Net solution the easiest way is to to build the Visual Studio solution which is how Visual Studio works too. So I pick ‘Visual Studio (sln)’ for the ‘Runner type’:



The important settings for this step are:

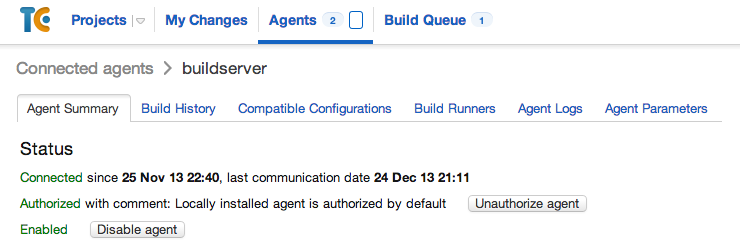
* Step name: like I said before pick a right name so it’s easy to figure out what the step does. I think ‘1. Build Solution’ is quite nice.
* Execute step: I leave this as default - ‘If all previous steps finished successfully’.
* Solution file path: you should tell TeamCity where the solution file is so it can build it. You can fill this easily by clicking on the tree icon next to the box and finding the solution file in the provided file browser as shown in the above screenshot.

I leave the rest as default.

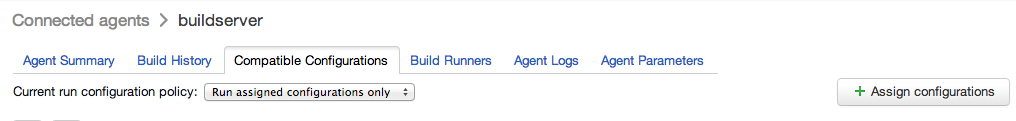
Assign your build configuration to an agent

Once you have a build step to build your source code, it’s a good idea to run your build to see if it works. For that to work though you have to assign a build agent to your build. So I am going to jump ahead a bit and explain how you can assign your build configuration to a build agent.

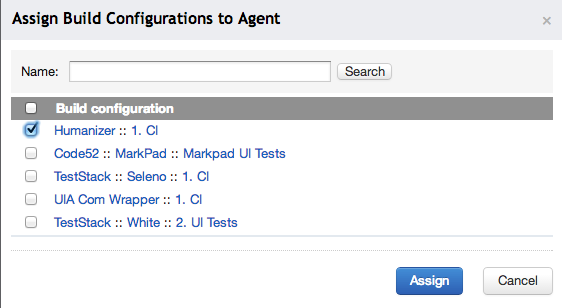
On the top navigation bar click on the ‘Agents’ link and then navigate to the build agent you want to use for your CI build configuration:



From there you go to the ‘Compatible Configurations’ pane where you can assign your new build configuration to your agent:



From there you just click ‘Assign configurations’, select the configuration you want to assign to this agent, which is your recently created build configuration, in my case Humanizer :: 1.CI:



Humanizer is the name of the TeamCity project and 1.CI is the name of my build configuration.

**Your first build run**

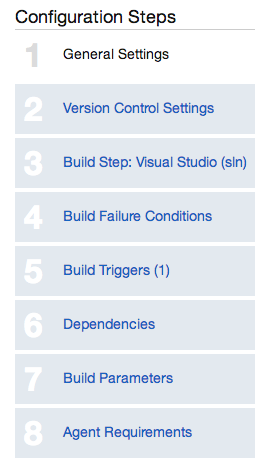
So now you have your build configuration with one build step and an agent assigned to it. Go forth and click run on your build. It should get the source from your repository and build it:



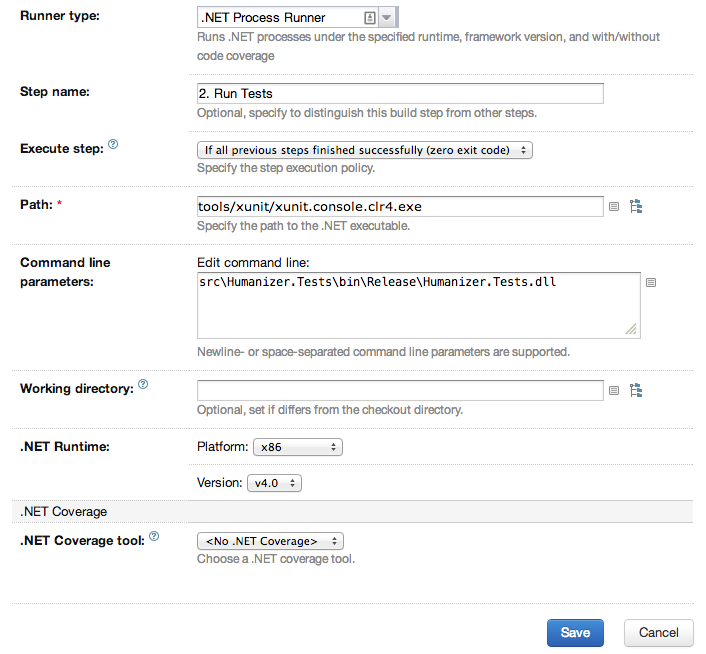
Make sure your build is green and it does what it should do: getting the latest code and building your solution. Check the ‘Changes’ and ‘Build Log’ panes.

**2. Run Tests**

Now that we have a passing build step, lets add test run to our build. To add a new build step, go to your build configuration and on the left navigation bar click on ‘3. Build Step(s)’:



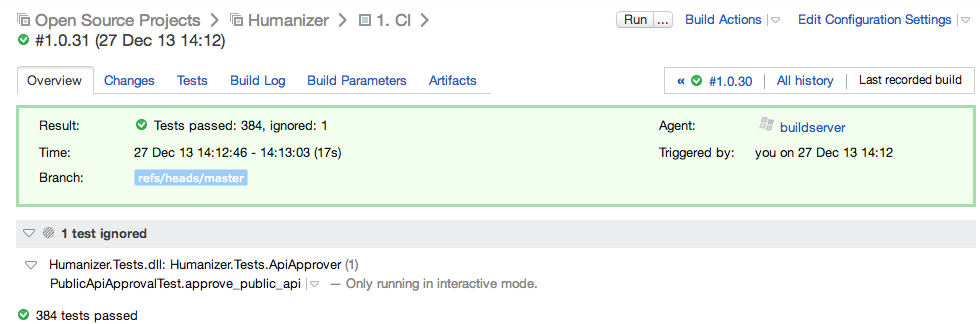
Now you can click ‘Add build step’ button to create your test runner step. As mentioned I am using xUnit for Humanizer and there isn’t a built-in xUnit runner in TeamCity unlike nUnit and MSTest. If you are using nUnit then you can just pick nUnit runner from the ‘Runner type’ dropdown and fill the settings pages (obviously the same applies to MSTest; but hopefully you’re not using MSTest). Before switching to TeamCity I had an MSBuild script which would do pretty much everything I am explaining here but it was run from command prompt. To run the tests from my build script I had to download xUnit and put it [in a ‘tools’ folder in my repository](https://github.com/MehdiK/Humanizer/tree/master/tools/xunit). So I just reused the xUnit in my repository to setup xUnit test run in TeamCity:



The important settings are:

* ‘Runner type’ is set to ‘.Net Process Runner’ to run the xUnit console runner.
* ‘Step name’ is set to ‘2. Run Tests’ for readability.
* ‘Path’ is the path to the .Net process I wish to run, in this case xunit.console.clr4.exe picked using the source tree explorer (the tree icon next to the Path box).
* ‘Command line parameters’: set to the relative path of my test dll: ‘src\Humanizer.Tests\bin\Release\Humanizer.Tests.dll’.

The rest are left as default values. Make sure you run your CI build again, this time with the test step:

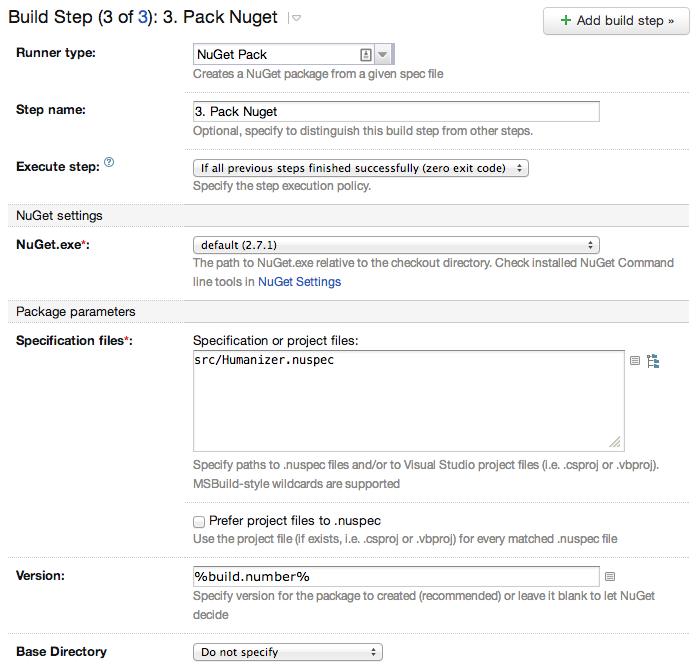


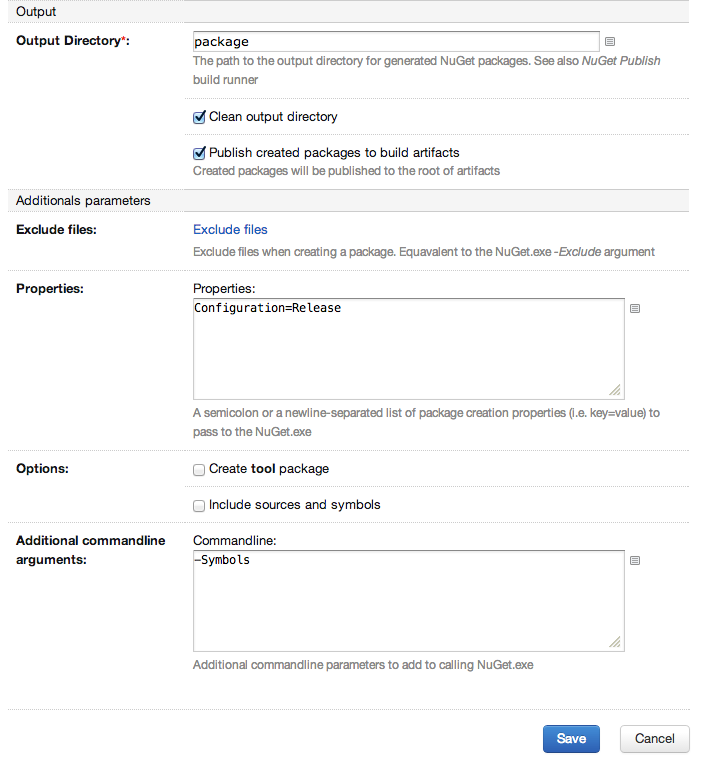
**Note the test results lighting up in the build result, oh yeah :)**

This setup works nicely for your unit and integration tests; but setting up TeamCity (or any CI server) to run UI tests is a bit more tricky. My friend Jake Ginnivan has an awesome post [here](http://jake.ginnivan.net/teamcity-ui-test-agent) about setting up UI tests on TeamCity that you must read if you do any UI testing.

**3. Pack NuGet**

A great practice in Continuous Delivery is to be able to publish the artifacts of any existing green build/test to production with a push of a button. In other words when you want to deploy, you don’t rebuild stuff - you just deploy the existing artifacts of a healthy build. We have now setup a build and test run in our CI. We should extract the deployment artifacts now so we can later use them for deployment. Humanizer is a .Net library and releasing this library means pushing a new NuGet package to [nuget.org](http://nuget.org/). So lets create the NuGet package from the build artifacts:





**The important settings are:**

* ‘Runner type’ is set ‘NuGet Pack’ to create the NuGet package.
* ‘Step name’ is set to ‘3. Pack NuGet’ for readability.
* ‘Specification files’ is set to the [existing ‘Humanizer.nuspec’ file](https://github.com/MehdiK/Humanizer/blob/master/src/Humanizer.nuspec). More on this shortly.
* In the ‘Output’ setting group, we tick the ‘Publish created packages to build artifacts’ checkbox to make sure the created NuGet packages end up in the CI build artifacts so we can later use them for deployment.
* ‘Additional commandline arguments’ is set to ‘-Symbols’. This tells NuGet to create a Symbols package to be published to [symbolsource.org](http://symbolsource.org/). More on this shortly.

Before we go ahead and run the build again I would like to highlight a few things on these settings. Lets take a look at my nuspec file first:

<?xml version="1.0"?>

<package >

<metadata>

<id>Humanizer</id>

<version>$version$</version>

<title>Humanizer</title>

<authors>Mehdi Khalili</authors>

<owners>Mehdi Khalili</owners>

<projectUrl>https://github.com/MehdiK/Humanizer</projectUrl>

<requireLicenseAcceptance>false</requireLicenseAcceptance>

<description>A framework that turns your normal strings, type names, enums and DateTime into a human friendly format and provides human friendly API for DateTime, TimeSpan etc</description>

<copyright>Copyright 2012-2013 Mehdi Khalili</copyright>

<licenseUrl>https://github.com/MehdiK/Humanizer/blob/master/LICENSE</licenseUrl>

<releaseNotes>

In this version we changed the library to Portable Class Library. There are no breaking changes.

</releaseNotes>

</metadata>

<files>

<file src="Humanizer\bin\Release\\*\*" target="lib\portable-win+net40+sl50+wp8" />

<file src="Humanizer\\*\*\\*.cs" exclude="\*\*\obj\\*\*\\*.\*" target="src" />

</files>

</package>

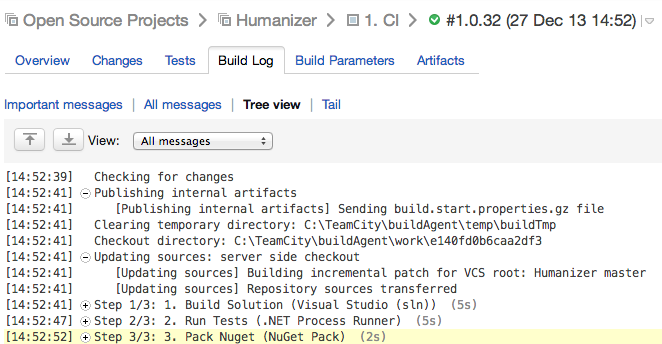
There are a few things of note in this file:

* The version element in metadata section is set to $version$: instead of specifying a static version number I want it injected by TeamCity build. The value injected into this element is1.0.%build.counter% as set in ‘build number format’ in CI build configuration before.
* <file src="Humanizer\bin\Release\\*\*" target="lib\portable-win+net40+sl50+wp8" /> tells nuget command to fetch all the files from ‘Humanizer\bin\Release', **which is relative to the nuspec file path**, and copy them to the ‘lib\portable-win+net40+sl50+wp8’ path relative to the NuGet output folder, which we set to ‘package’ in our NuGet step. portable-win+net40+sl50+wp8 means that the library is a Portable Class Library that can target Win Store Apps (*win*), .Net 4 & higher versions (*net40*), SilverLight 5 (*sl50*) and Windows Phone 8 (*wp8*). To see a complete list of values for nuspec for Portable Class Library please check [this article](http://nuget.codeplex.com/workitem/1228).
* <file src="Humanizer\\*\*\\*.cs" exclude="\*\*\obj\\*\*\\*.\*" target="src" /> tells NuGet command that we also want the source files to be included in the ‘src’ target folders!!! But this is a library - why would we need source files?! The answer is to create a symbols NuGet package to publish to [symbolsource.org](http://www.mehdi-khalili.com/continuous-integration-delivery-github-teamcity/'http://symbolsource.org/') so the users of the library can step through the code while debugging.

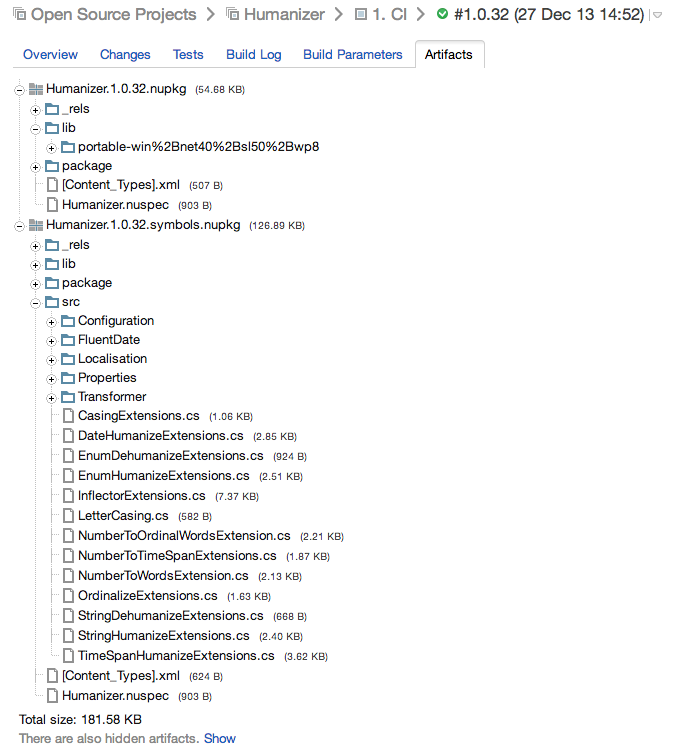
You can read [this article](http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-symbol-package) to learn more about how you can create a Symbol Package. The gist of that is:

* Copy all your source files to a ‘src’ folder in your NuGet folder (which we’re doing using the ‘file’ element in the ‘files’ section of the nuspec file).
* Call nuget pack with a -Symbols parameter to create an additional Symbols package which we are doing by setting ‘Additional commandline arguments’ in ‘3. Pack Nuget’ build step to ‘-Symbols’.

Alright lets run the CI build and see what we get:



We can see the three build steps in the build log. Also if we check the ‘Artifacts’ pane we can see the NuGet packages:



And in the build log we can see that the version has been injected into nuspec:

Attempting to build package from 'Humanizer.nuspec'.

Id: Humanizer

Version: 1.0.34

Authors: Mehdi Khalili

Description: A framework that turns your normal strings, type names, enums and DateTime into a human friendly format and provides human friendly API for DateTime, TimeSpan etc

License Url: https://github.com/MehdiK/Humanizer/blob/master/LICENSE

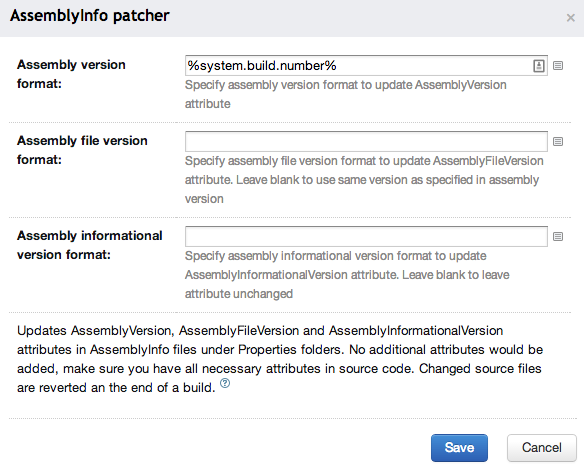
Project Url: https://github.com/MehdiK/Humanizer

Dependencies: None

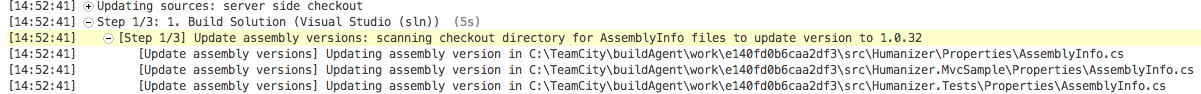
**Rewriting Assembly Versions**

Although our CI build configuration can build the code, run the tests and package the artifacts it has a relatively big flaw, and that is the library version as stored in [‘AssemblyInfo.cs’](https://github.com/MehdiK/Humanizer/blob/master/src/Humanizer/Properties/AssemblyInfo.cs) files is static and while the published NuGet package version increases, the version of the dll inside the package never changes as it’s set to a static value!

TeamCity has a very elegant solution for this problem: ‘AssemblyInfo patcher’. In the ‘Build Steps’ page of your Build Configuration there is a section down the bottom called ‘Additional Build Features’ where you can, errrrm, add additional features to your build! Click on the ‘Add build feature’ button and add ‘AssemblyInfo patcher’ feature:



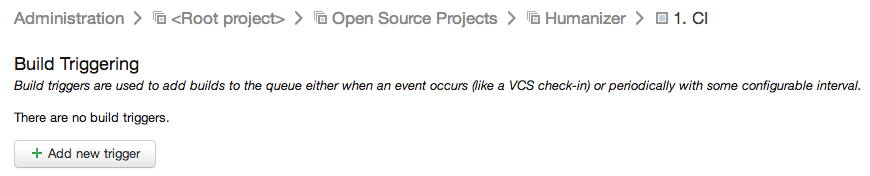
That’s it!! TeamCity takes care of everything for you. When you run your build again you can see the magic happening in your build log:



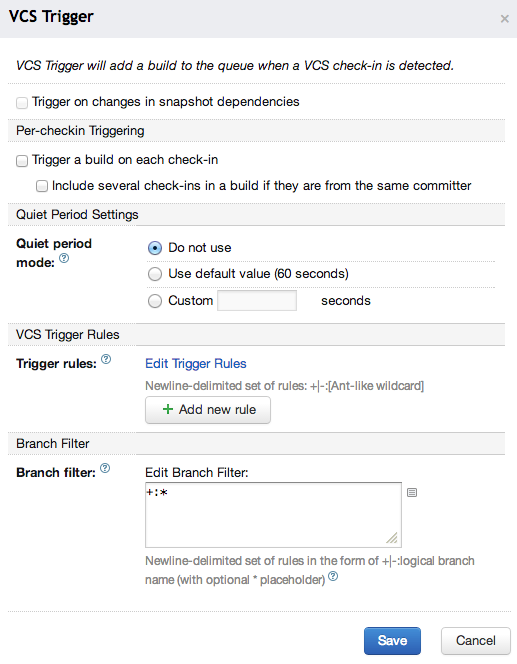
Another thing of note is that this expectedly happens as part of your ‘1. Build Solution’ build step so the ‘3. Pack Nuget’ step packs the build artifacts with correct versions.

**Build Trigger**

We now have a complete CI build configuration. There is still one problem though: we have been running this build configuration manually! A CI setup should be able to detect changes on the source code and build them automatically. We can achieve that using ‘Build Triggers’. Again from the left navigation bar on your CI Build Configuration select ‘5 Build Trigger’ and click on ‘Add new trigger’ button to add a build trigger.



To make the trigger dependent on source code changes we choose ‘VCS Trigger’:



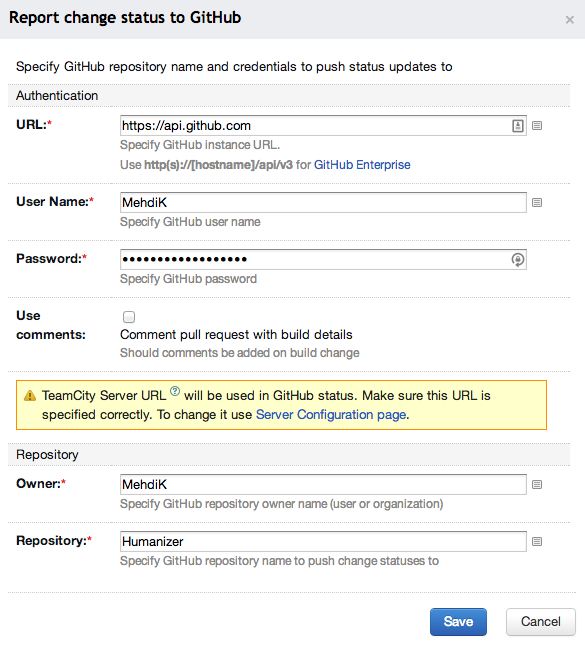
Now you have automatic builds on all branches of your source control including pull requests.

**Pull Request Notification**

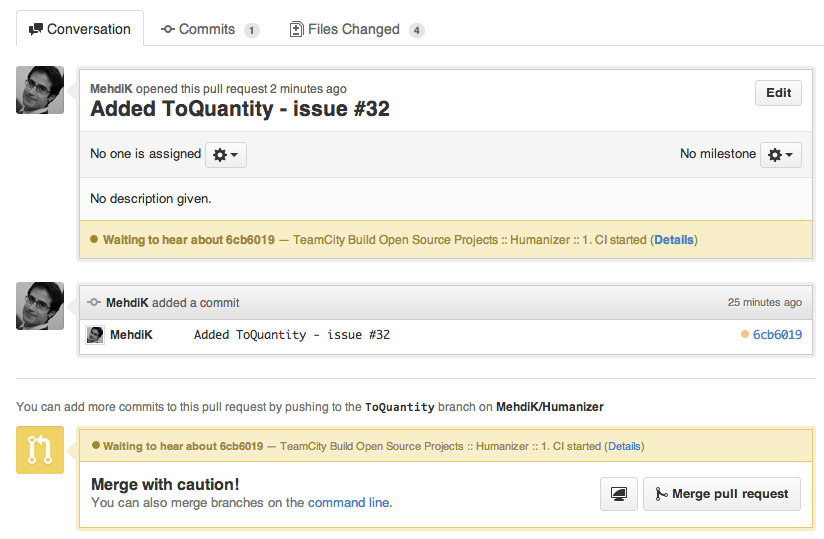
TeamCity can automatically build GitHub pull requests and provide build notifications on them which is pretty handy. Hadi Hariri has a detailed post about the feature [here](http://hadihariri.com/2013/02/06/automatically-building-pull-requests-from-github-with-teamcity/).

Basically you have a TeamCity CI build setup for your GitHub project which builds your code and runs your tests on checkin. Why not leverage that to provide notification to contributors of your project?

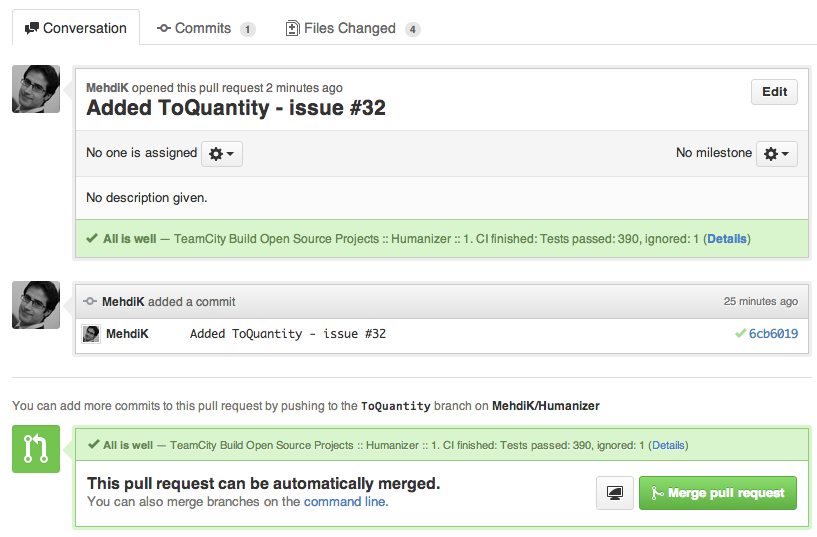
We have everything setup for that. All we need now is to add the notification in our configuration and see the result light up in GitHub pull requests. Go to your ‘Build Steps’ page and add a new ‘Report change status to GitHub’ build feature:



All settings are quite straightforward and now the notification:



You see a pending notification while TeamCity is building the PR code and when the build is finished you get the result on your PR page, in this case a successful build:



Showing build status icon on GitHub

We can get a [build status icon](http://blog.jetbrains.com/teamcity/2012/07/teamcity-build-status-icon/) which is quite handy for GitHub read me page. To achieve this add the following snippet to your html page:

<a href="http://teamcity/viewType.html?buildTypeId=btN&guest=1">

<img src="http://teamcity/app/rest/builds/buildType:(id:btN)/statusIcon"/>

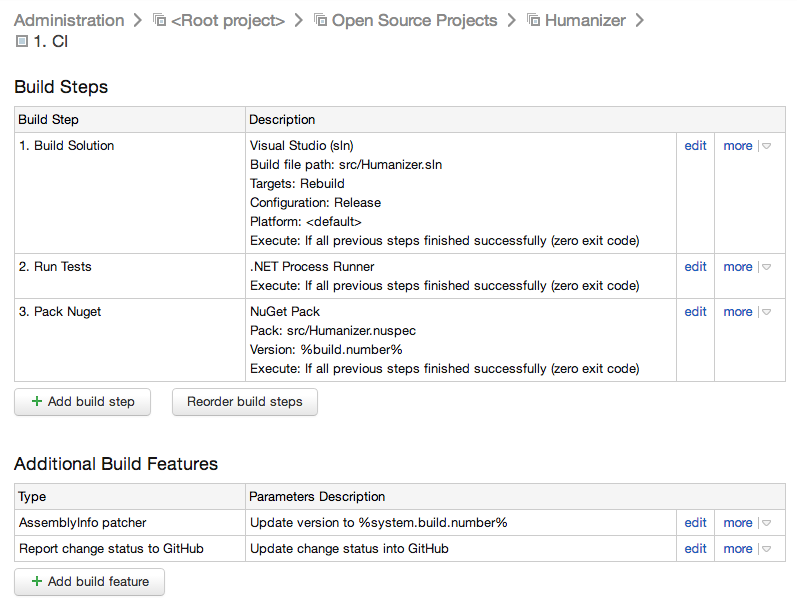
</a>

Remember the ‘Build configuration ID’ we set on ‘Build Configuration’ (in my case Humanizer\_CI). You should replace the build type Ids (btN) on the above snippet with your ‘Build Configuration Id’, and obviously teamcity should be replaced with your TeamCity server URL.

You can check my build icon on Humanizer’s [ReadMe page](https://github.com/MehdiK/Humanizer#continuous-integration-from-teamcity). This icon is also a link to the build page and clicking on it takes you to the TeamCity project where you can see the build history, click on each build entry and see the logs, tests, artifacts etc. You see that guest=1 in the query string?! That means that you don’t have to be a TeamCity user to be able to see the build: you will be logged in as a guest user with limited view only access.

**Wrapping up the CI build**

To sum up our CI build, we created a build configuration, called ‘1. CI’, under our TeamCity project. This is what my build steps page look like now:



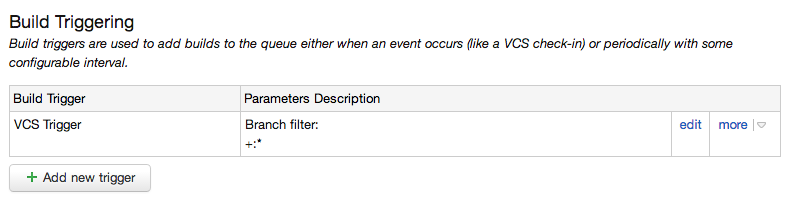
**‘1. CI’ build configuration has three build steps as seen in the screenshot:**

* ‘1. Build Solution’
* ‘2. Run Tests’
* ‘3. Pack NuGet’

And two build features:

* ‘AssemblyInfo patcher’
* ‘Report change status to GitHub’

I also have one VCS build trigger:



**Setting up Continuous Delivery**

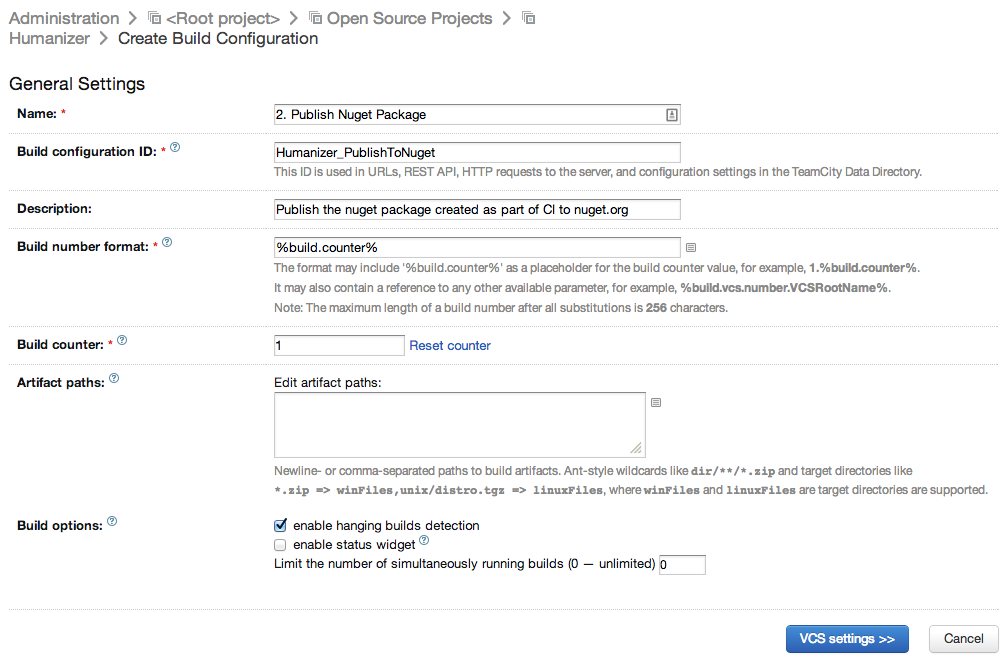
Now that we have CI setup nicely, lets setup [Continuous Delivery](http://martinfowler.com/bliki/ContinuousDelivery.html) for our project:

*Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.*

*The key test is that a business sponsor could request that the current development version of the software can be deployed into production at a moment’s notice - and nobody would bat an eyelid, let alone panic.*

I need a separate build configuration because I don’t want to deploy every single successful build to production (in my case NuGet) and you shouldn’t either. I create a separate build configuration so I can run it whenever I want on a successful build I am happy with.

To create our CD build we go to our project home page and click on ‘Create build configuration’:



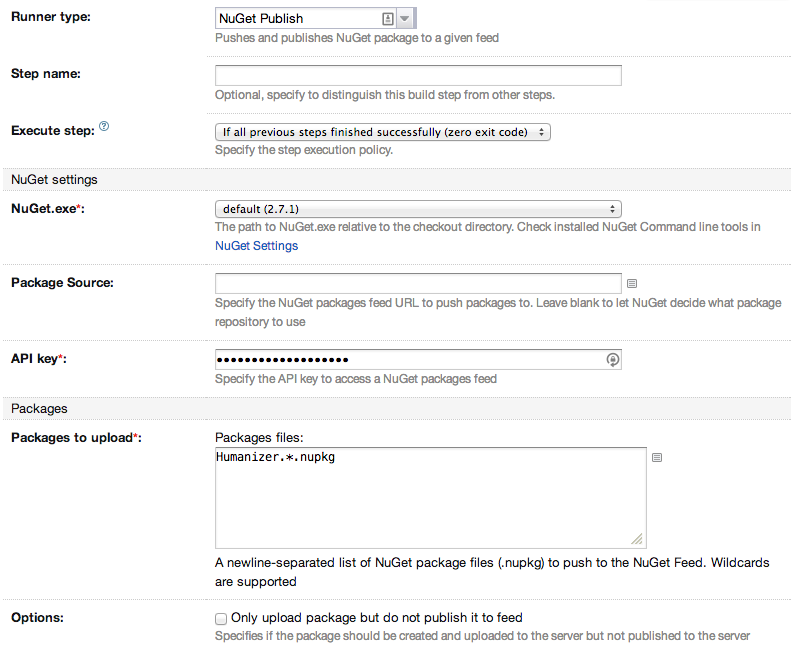
Nothing of note here. Just make sure you give your build some name and description.

**Source Control Settings**

Back in ‘Pack Nuget’ I briefly talked about Continuous Delivery which I would like to repeat here: *“A great practice in Continuous Delivery is to be able to publish the artifacts of any existing green build/test to production with a push of a button. In other words when you want to deploy, you don’t rebuild stuff - you just deploy the existing artifacts of a healthy build..”*. We already have a build step, 3. Pack Nuget, that creates the artifacts required for deployment. So I don’t need any version control settings for this build (and you shouldn’t either) because your CD build should work off the existing build artifacts created by your CI build with no dependency on any code. So we skip over ‘Version Control Settings’.

**Publish NuGet Package Build Step**

We should add a build step to publish our artifacts to production. In my case I want to publish my NuGet package(s):



**Things of note in this step are:**

* ‘Runner type’ is set to ‘NuGet Publish’ which is what I want to do.
* I haven’t given this build step a name, mainly because my CD build has one and only one step. When you have only one step, name is not that necessary, as we will see later.
* ‘API Key’ is your NuGet API key which you can get from your [NuGet account page](https://www.nuget.org/account). Just grab that GUID and paste it here.
* ‘Packages to upload’ is set to ‘Humanizer.\*.nupkg’ where Humanizer is the id of my NuGet package. The \* is to match all NuGet packages and versions; e.g. ‘Humanizer.1.0.34.nupkg’ and ‘Humanizer.1.0.34.symbols.nupkg’.

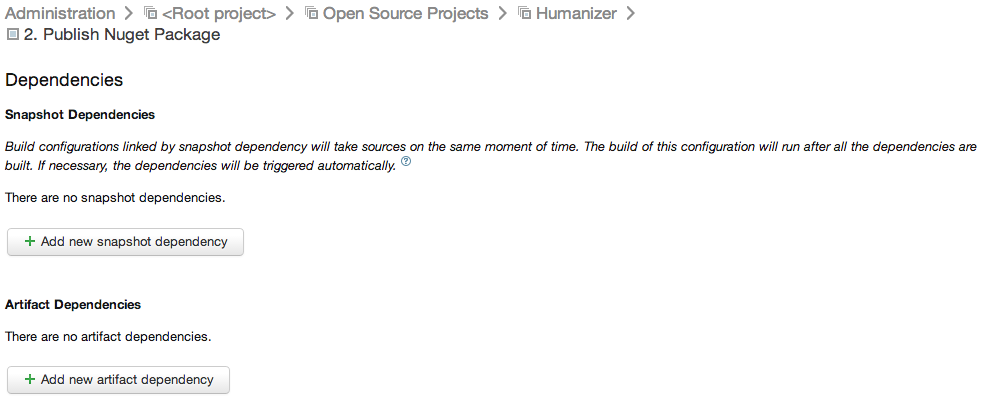
If you only want to deploy your package but not list them you can tick ‘Only upload package but do not publish it to feed’.

**Build Triggers**

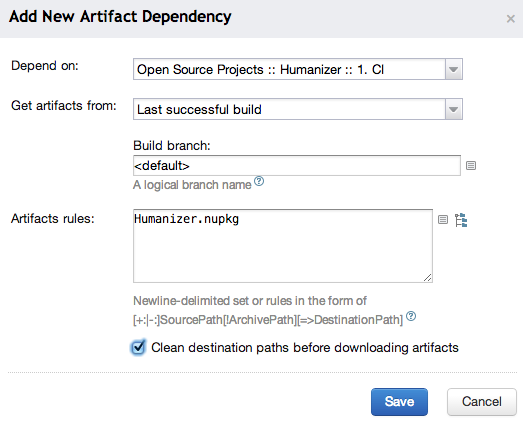
Unlike our CI build, our CD build configuration doesn’t need a build trigger because IMO it shouldn’t happen automatically. I want a CD build run to be a manual process so I can deploy to production any time I want. So I leave this empty.

**Getting deployable artifacts**

The ‘Publish NuGet Package’ build step depends on ‘Humanizer.\*.nupkg’ but doesn’t know where to get it from; so we have to somehow resolve it. We can do that using build ‘Dependencies’. Click on the ‘Dependencies’ on the left navigation bar:

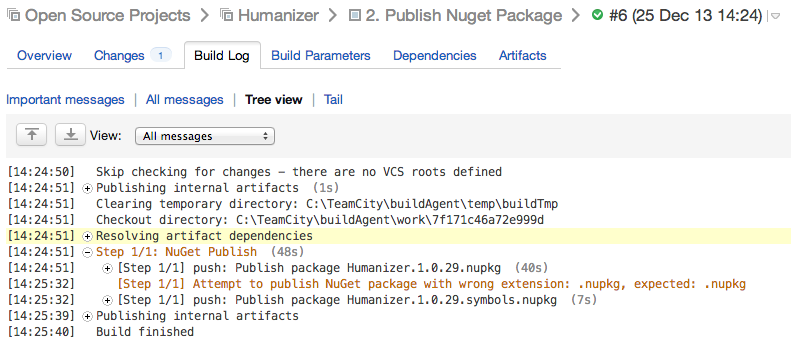


Now ‘Add a new artifact dependency’:



From the ‘Depend on’ box you can pick the artifacts we published in the ‘Pack NuGet’ step by ticking ‘Publish created packages to build artifacts’ checkbox. Back in the dependencies page, there is a button down the bottom called ‘Check artifact dependencies’ which helps you verify the dependencies you have specified. It’s a good idea to check your dependencies now.

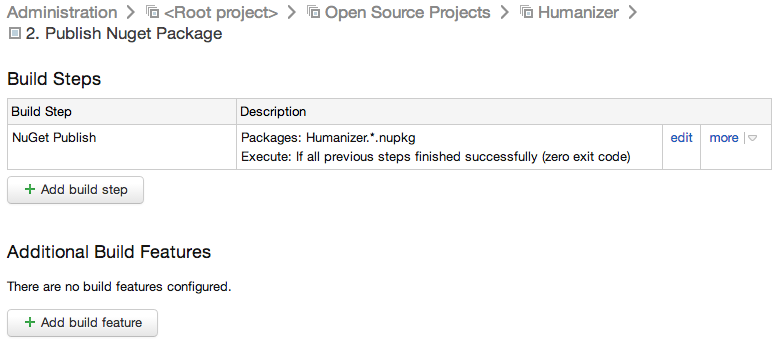
You can now run your CD build and see your build artifacts pushed to production:



We used ‘NuGet Publish’ on ‘Humanizer.\*.nupkg’ in our only build step; but since we had -Symbols in our ‘Pack NuGet’ step we had two NuGet packages and in the log we can see that both packages have been deployed: one to nuget.org, which we can see [here](https://www.nuget.org/packages/humanizer), and another one to symbolsource.org, which we can see [here](http://www.symbolsource.org/Public/Metadata/NuGet/Project/Humanizer).

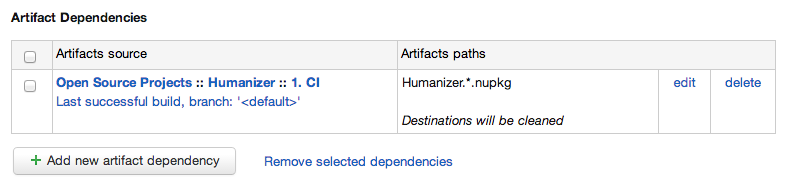
**Wrapping up the CD build**

To sum up our CD build, we created a build configuration, called ‘2. Publish NuGet Package’ (which in the hindsight I should’ve called ‘2. CD’), under our TeamCity project. This is what my build steps page looks like now:



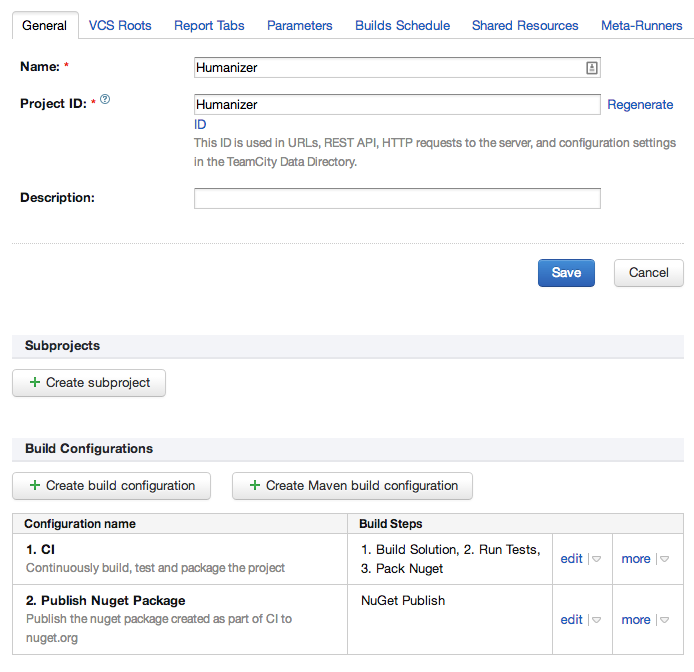
Our CD build has one step only: ‘NuGet Publish’. Even though we didn’t give this step a name, TeamCity shows a name based on the Runner type. That’s why I didn’t give this step any name. Still not a bad idea to have a name though.

We also have an artifact dependency:



Conclusion

In this post we created a TeamCity project and setup continuous integration and delivery for a .Net project hosted on GitHub along with automatic CI run on code checkins and pull requests. Here is the 1000-foot view of our project:



Hopefully this post provides an easy guide for setting up your TeamCity projects. If you are a TeamCity or CI/CD expert and think I could/should have done something different then I would love to hear your thoughts. Please leave me a comment.

P.S. Thanks Jake for letting me use your server.