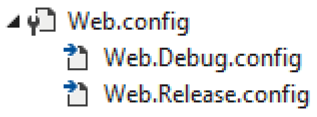
Continuous Delivery to Windows Azure Web Sites (or IIS)

In this tutorial, we'll go over the basics of these and see how we can deploy an ASP.NET MVC project to IIS or Windows Azure Web Sites from our TeamCity server using WebDeploy.

Deploying ASP.NET applications can be done in a multitude of ways. Some build the application on a workstation and then xcopy it over to the target server. Some use a build server, download the artifacts, change the configuration files and xcopy those over to the server. The issue with that arises when something bad creeps in: deployments become unpredictable. What if there are leftovers of unnecessary or old assemblies on that workstation we're xcopying from? What if we forget to change the database connection string in Web.config and mess up that release? How do we quickly roll back if that happens? The .NET stack has a solution to this: Configuration Transforms and WebDeploy.

Configuration Transforms



One of the things that typically have to happen during deployment is making changes to the configuration. Changing the database connection string, changing ASP.NET settings to no longer show us YSOD's and so on. Don't hard-code these things or write a big if-else statement based on the server's hostname to figure out the configuration. Instead, use something like configuration transforms.

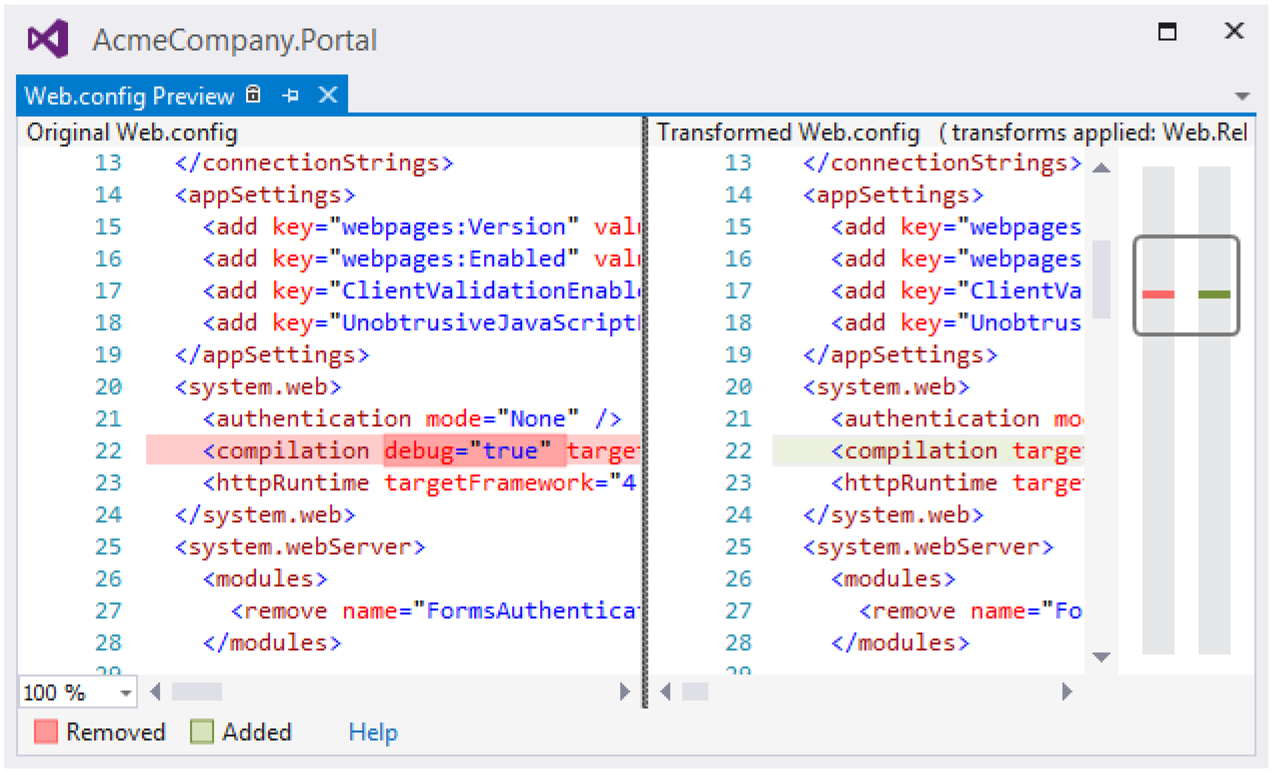
Configuration transforms are files that describe "transformations" to Web.config, based on the build configuration being used. Building the Release configuration? Then Web.config will be updated with the rules described in Web.Release.config. Let's remove the debug attribute from our configuration when doing a Release build:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <system.web>
    <compilation xdt:Transform="RemoveAttributes(debug)" />
  </system.web>
</configuration>
```

A typical ASP.NET application created in Visual Studio will contain transforms for Debug and Release builds, but they can be added by creating a new build configuration (through the Build | Configuration Manager… menu) and then using the context menu Add Config Transform.

For this tutorial, I've created 2 new configurations: Development and Production, and generated 2 new configuration transforms as well (Web.Development.config and Web.Production.config).

To test the config transform, we can make use of the context menu Preview Transform, which will show us exactly what the resulting configuration file is going to look like. The following is the result of running the Web.Release.config transform:


AcmeCompany.Portal

Web.config Preview

Original Web.config

```
</connectionStrings>
<appSettings>
  <add key="webpages:Version" val
  <add key="webpages:Enabled" val
  <add key="ClientValidationEnabl
  <add key="UnobtrusiveJavaScript
</appSettings>
<system.web>
  <authentication mode="None" />
  <compilation debug="true" targe
  <httpRuntime targetFramework="4
</system.web>
<system.webServer>
  <modules>
    <remove name="FormsAuthentica
  </modules>
```

Transformed Web.config (transforms applied: Web.Rel

```
</connectionStrings>
<appSettings>
  <add key="webpages
  <add key="webpages
  <add key="ClientVa
  <add key="Unobtrus
</appSettings>
<system.web>
  <authentication mo
  <compilation targe
  <httpRuntime targe
</system.web>
<system.webServer>
  <modules>
    <remove name="Fo
  </modules>
```

100 %

🟥 Removed 🟩 Added Help

We can use this to virtually change or add any setting we'd like to change. Connection strings, file paths, app settings, diagnostics configuration and so on. Here's some more documentation on what you can do with config transforms.

WebDeploy

For several versions, Visual Studio has had the option to create so-called "web packages" for any ASP.NET application, containing all files required to run the app. Pages, images, CSS, JavaScript and the application binaries can be exported in such package. It's even possible to include databases and IIS settings!

These deployment packages can be used together with WebDeploy, a tool which can upload the package to a server using various protocols and can apply the config transforms we've talked about earlier.

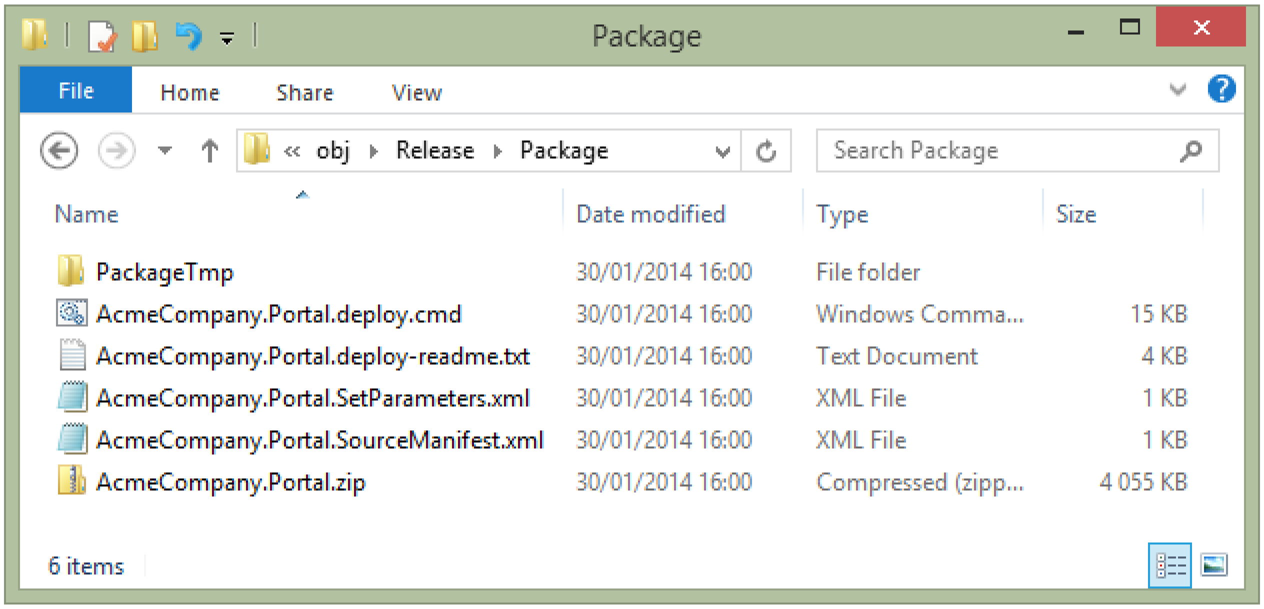
But before we deploy, let's first see how we can create a deployment package. And just so we learn about the package format, let's first do this manually by invoking msbuild.

Manually creating a deployment package

Deployment packages can be created by running the Package build target on the project, which can easily be done using msbuild:

```
msbuild AcmeCompany.Portal.csproj /T:Package /P:Configuration=Release
```

The project will be compiled and a new folder created, containing our deployment package. And more!



Name	Date modified	Type	Size
PackageTmp	30/01/2014 16:00	File folder	
AcmeCompany.Portal.deploy.cmd	30/01/2014 16:00	Windows Comma...	15 KB
AcmeCompany.Portal.deploy-readme.txt	30/01/2014 16:00	Text Document	4 KB
AcmeCompany.Portal.SetParameters.xml	30/01/2014 16:00	XML File	1 KB
AcmeCompany.Portal.SourceManifest.xml	30/01/2014 16:00	XML File	1 KB
AcmeCompany.Portal.zip	30/01/2014 16:00	Compressed (zipp...	4 055 KB

6 items

The ZIP file contains our application, the other files are supporting files for deploying to a target machine. An interesting file is AcmeCompany.Portal.SetParameters.xml. It contains the result of our config transforms, but allows for overriding these values. Why? Well, the person building the deployment package may not know the connection string. Imagine only an administrator knows? That person can override the setting with the correct, final connection string for production through this file.

The AcmeCompany.Portal.deploy.cmd batch file can be run to deploy to a target environment, but... how does that work?

WebDeploy can make use of several methods to transfer the deployment package to a remote server and update configuration. It can be done using WebDeploy (an HTTPS based protocol), FTP or using a File Share. For the first option, some additional tools should be enabled on the target IIS server. With good reason: the WebDeploy server-side tool will do real synchronization between sites and delete redundant content from the server. For FTP or a file share, no additional tools are required.

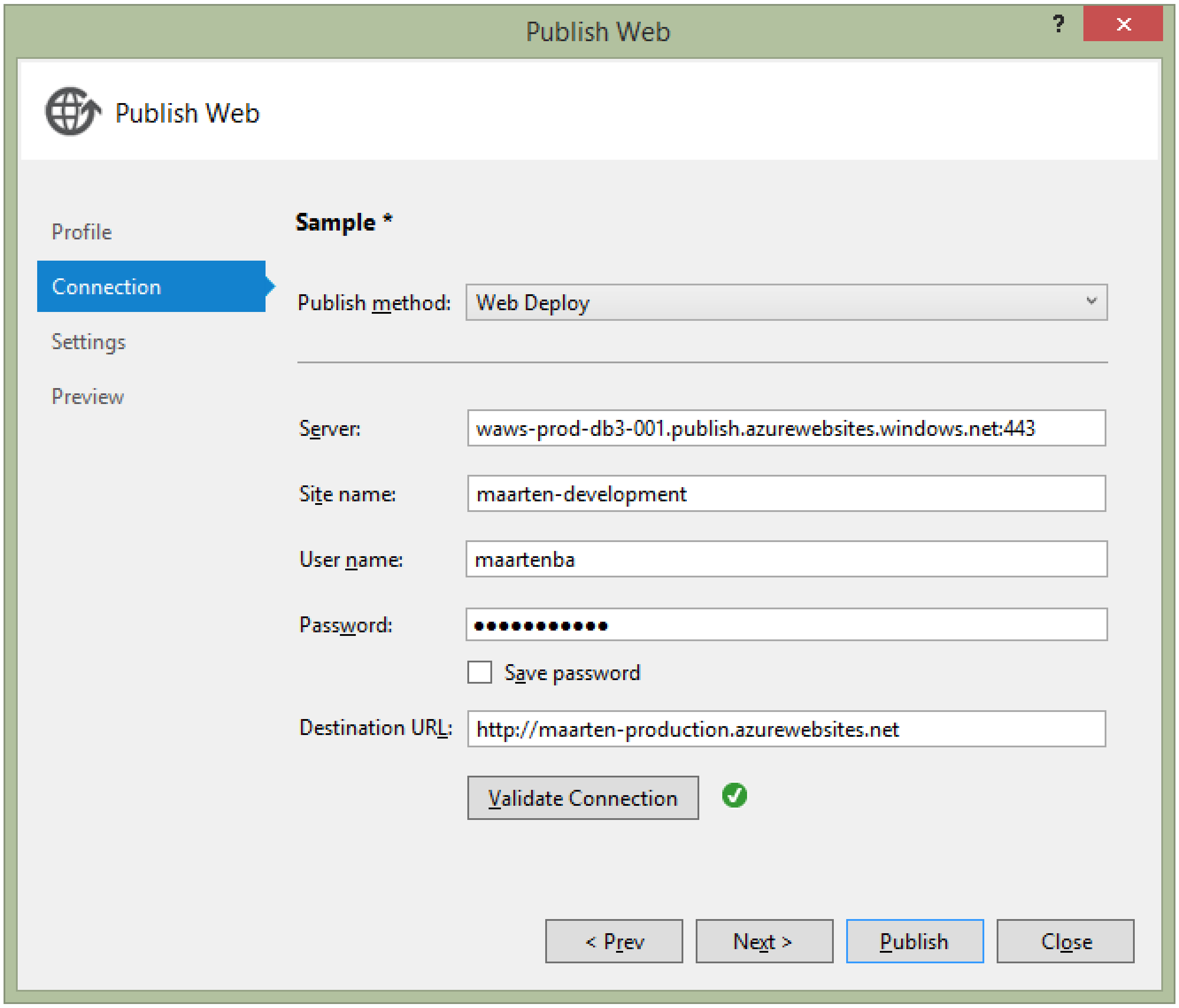
For the remainder of this tutorial, we will be covering deployment to Windows Azure Web Sites using WebDeploy, which is identical to how it works on IIS.

Step 1: Configuring deployment packages / WebDeploy with Visual Studio

In the previous step, we've created a deployment package manually and we would also have to invoke WebDeploy manually. There is an easier way though: configuring deployment packages and WebDeploy in one go, from Visual Studio.

From the web application that should be deployed, use the context menu on the project node and click Publish. This will open up a dialog where we can do some configuration related to our deployment. We can even create multiple deployment profiles, for example one for staging and one for production.

In the first step, we have to specify destination server details. This would typically be the HTTPS endpoint to the WebDeploy host (or FTP or file share details if that option was selected). After providing all details, we can validate the connection to see if it works.


Publish Web

Sample *

Publish method: Web Deploy

Server: waws-prod-db3-001.publish.azurewebsites.windows.net:443

Site name: maarten-development

User name: maartenba

Password: ●●●●●●●●●●●●

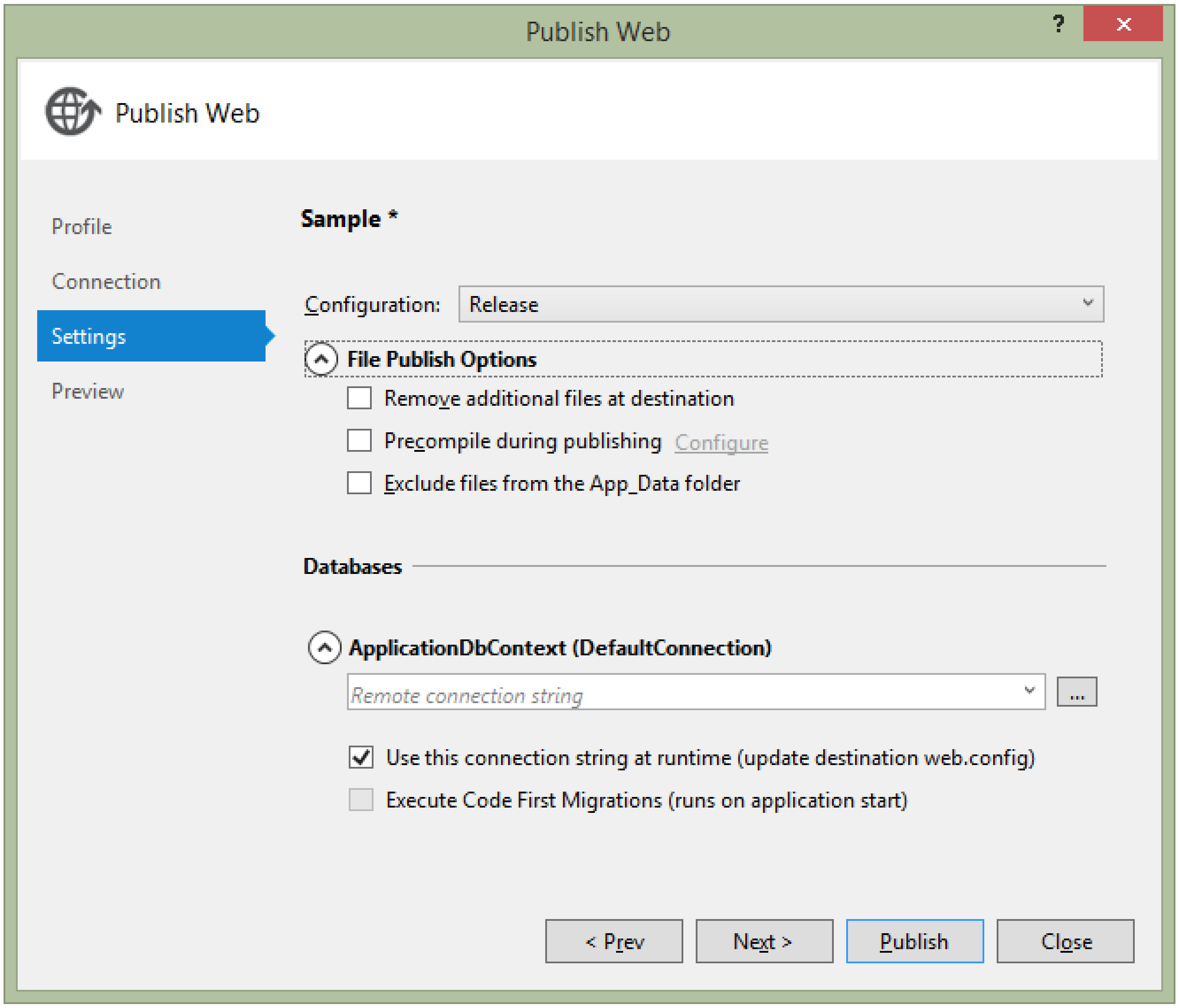
[ ] Save password

Destination URL: http://maarten-production.azurewebsites.net

Note that instead of going through the entire wizard, Windows Azure Web Sites tooling allows importing the publish profile from the Windows Azure Management Portal. I'm showing the entire process here for when deploying to IIS.

Does a password have to be specified? No! In case the developer doesn't know it, credentials can be left blank; we'll provide the username and password later on when deploying from TeamCity.

In the next step, we can specify some deployment specifics: should files that are not in the deployment package be deleted from the target server? Should the application be precompiled? Should the database connection string be overridden? And when using Entity Framework Code First: should migrations be executed?



Sample *

Configuration: Release

File Publish Options

- [ ] Remove additional files at destination
- [ ] Precompile during publishing Configure
- [ ] Exclude files from the App_Data folder

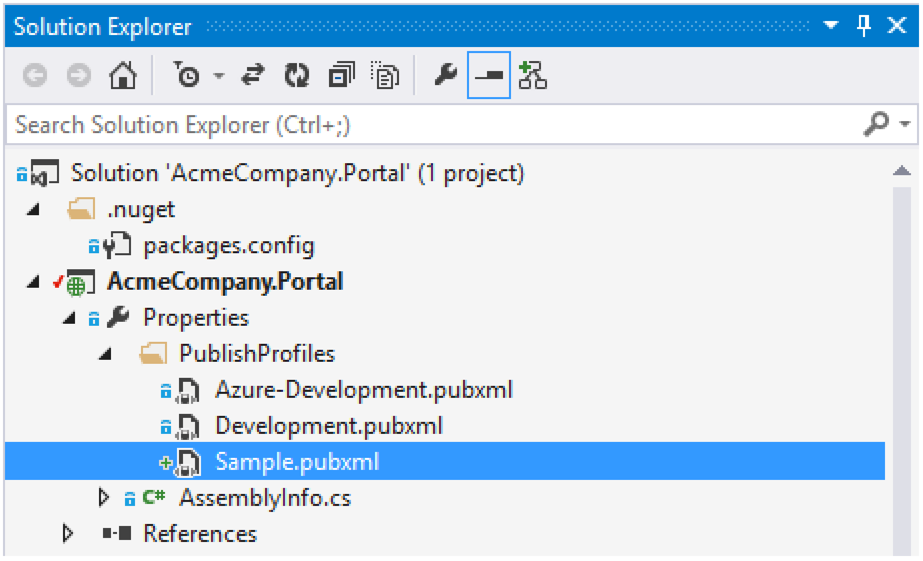
Databases

ApplicationDbContext (DefaultConnection)

Remote connection string

- [✓] Use this connection string at runtime (update destination web.config)
- [ ] Execute Code First Migrations (runs on application start)

We can close the wizard after this step, and save the publish settings just created into a file in our project:



```
Solution 'AcmeCompany.Portal' (1 project)
  .nuget
    packages.config
  AcmeCompany.Portal
    Properties
      PublishProfiles
        Azure-Development.pubxml
        Development.pubxml
        Sample.pubxml
      AssemblyInfo.cs
    References
```

This is just an XML file and we can edit it if needed. And actually we should, to make our life easier later on. Open the XML file and find the <DesktopBuildPackageLocation> element. When running the WebDeploy packaging step from the command line (which TeamCity will effectively do), this location will not be found. To resolve this, change the element value and prefix the path with _$(SolutionDir)_. Here's an example of what this element could look like:

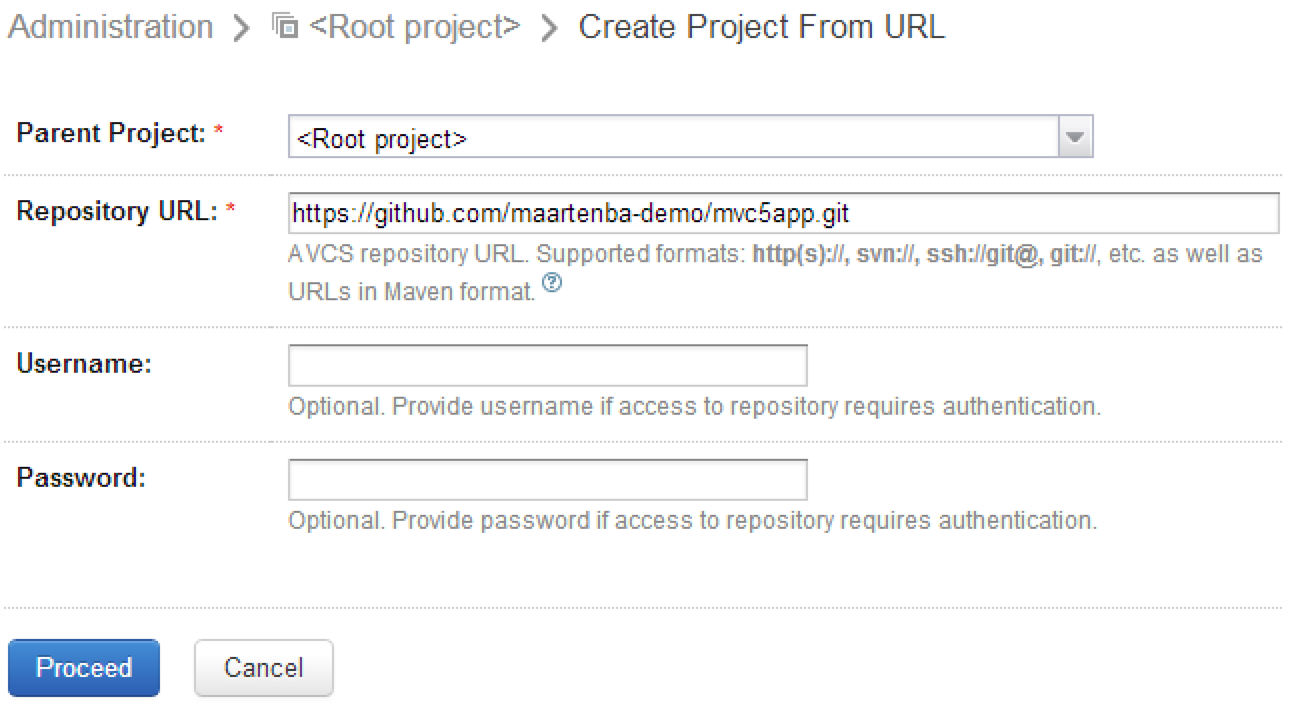
```
<DesktopBuildPackageLocation>$(SolutionDir)\artifacts\webdeploy\Development\AcmeCompany.Porta
l.zip</DesktopBuildPackageLocation>
```

Save the file and make sure it is added to source control so we can make use of it when running the deployment on TeamCity.

Step 2: Setting up the continuous integration build on TeamCity

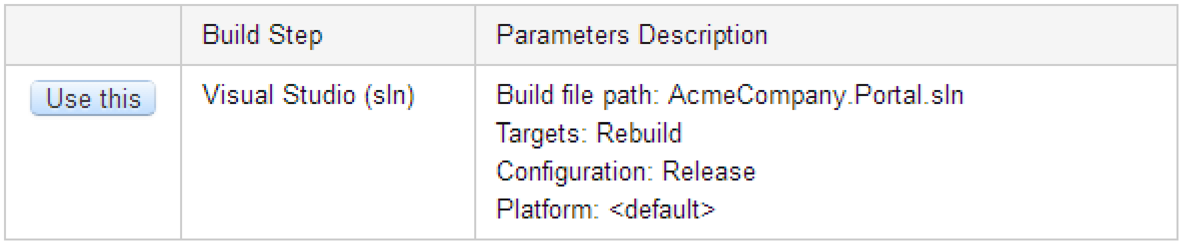
We want to have a continuous integration (CI) build for our project, which we can trigger on every VCS check-in. This CI build will provide us with immediate feedback on the project's build status and health.

TeamCity 8.1 allows us to create a project based on a VCS URL. We can simply enter the URL to a git, Mercurial, Subversion, ... repository:



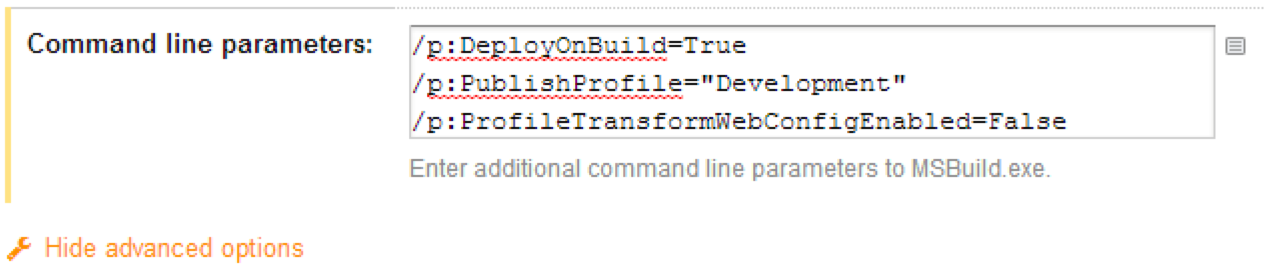
Parent Project: *	<Root project>
Repository URL: *	https://github.com/maartenba-demo/mvc5app.git A VCS repository URL. Supported formats: **http(s)://, svn://, ssh://git@, git://**, etc. as well as URLs in Maven format.
Username:	Optional. Provide username if access to repository requires authentication.
Password:	Optional. Provide password if access to repository requires authentication.

This repository will be analyzed and scanned for build steps. In our case, TeamCity discovered a Visual Studio 2013 build step which we can immediately add to our build configuration:



	Build Step	Parameters Description
Use this	Visual Studio (sln)	Build file path: AcmeCompany.Portal.sln Targets: Rebuild Configuration: Release Platform: <default>

Adding the suggested build step will result in a working build if we run it. We can specify artifact paths, version number and so on. One thing is missing though! The WebDeploy deployment package is nowhere to be seen. The reason for this is we are building the Rebuild target, which simply rebuilds our project without packaging. To solve this, we can add some additional command line parameters to our build step:



Command line parameters:

```
/p:DeployOnBuild=True
/p:PublishProfile="Development"
/p:ProfileTransformWebConfigEnabled=False
```

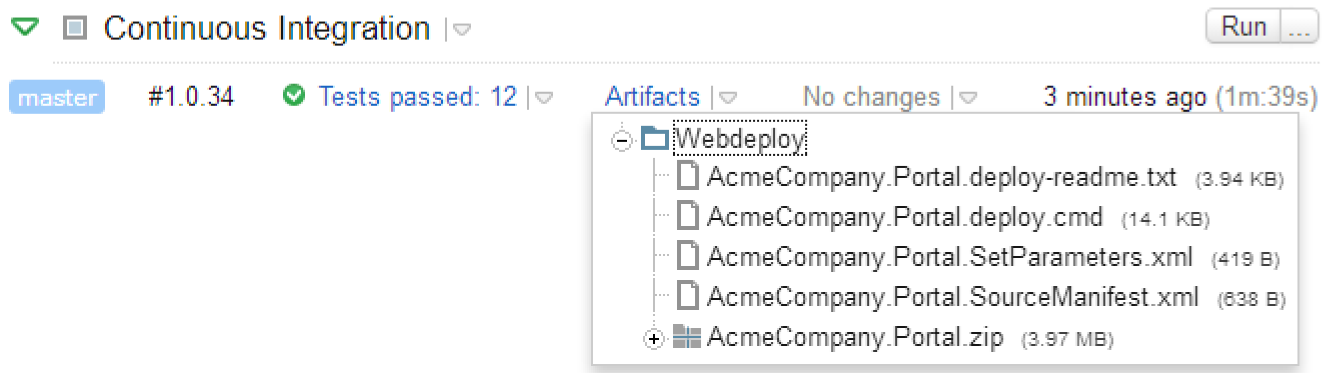
Enter additional command line parameters to MSBuild.exe.

Here's what these parameters do:

- /p:DeployOnBuild=True - triggers WebDeploy packaging
- /p:PublishProfile="Development" - specifies the deployment profile to use when packaging
- /p:ProfileTransformWebConfigEnabled=False - let's discuss this one in detail!

Standard configuration transformations are run in an early stage, but WebDeploy runs another transformation using the <LastUsedBuildConfiguration> setting from our publish profile. This causes earlier configuration transformations to be overwritten, which we don't want to happen. Disabling the ProfileTransformWebConfigEnabled parameter avoids running this additional configuration transformation.

If we now run the build again (having specified artifacts\webdeploy\Development => Webdeploy as the artifact path, which is the path we configured in the publish profile earlier on), we will see a familiar set of files published as artifacts:



```
Webdeploy
  AcmeCompany.Portal.deploy-readme.txt (3.94 KB)
  AcmeCompany.Portal.deploy.cmd (14.1 KB)
  AcmeCompany.Portal.SetParameters.xml (419 B)
  AcmeCompany.Portal.SourceManifest.xml (638 B)
  AcmeCompany.Portal.zip (3.97 MB)
```

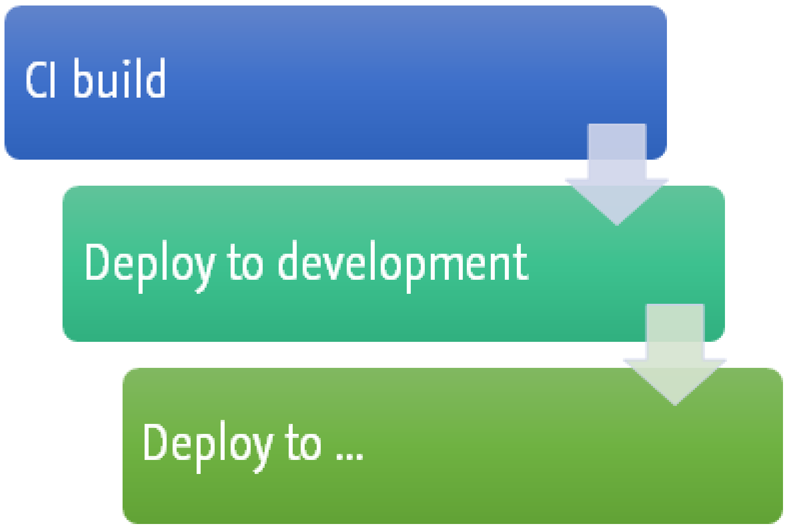
Now let's see if we can set up the actual deployment as well!

Step 3: Setting up the deployment on TeamCity

The strategy we'll be using for our deployments is described in the How To.... We will be creating a new build configuration for every target environment we want to deploy to. These new build configurations will:

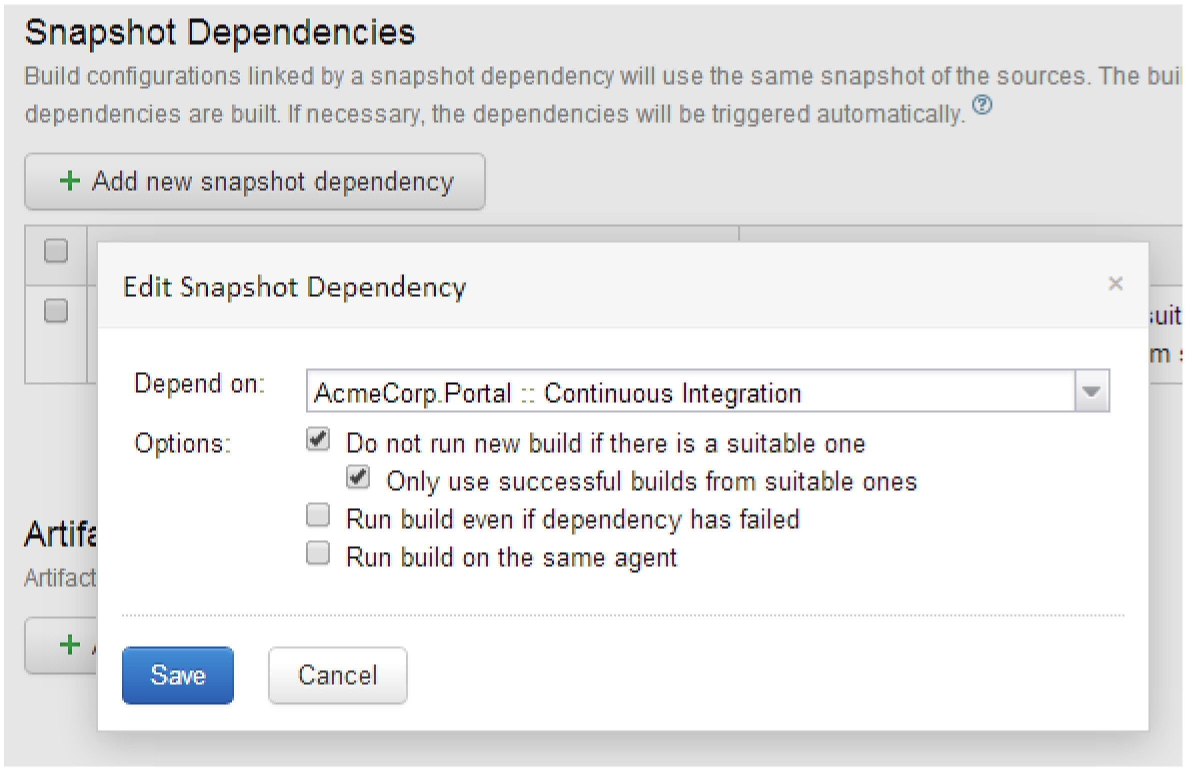
- Run the build
- Perform the deployment

What we want to achieve is this nice waterfall, where we can promote our build from CI to development to staging to production, or whichever environments we have in between CI and production.



From the TeamCity Administration, copy the CI build configuration and name it differently, for example “Deploy to Windows Azure Web Sites - Development”. Next, we will make some changes to the build configuration.

Let's start by specifying build dependencies. Under the build configuration's Dependencies, add a new snapshot dependency on our CI build. This will ensure that deployment will only be possible if a matching CI build has passed completely, and that the deployment will be based on the exact same VCS revision as we built during CI.



Snapshot Dependencies

Build configurations linked by a snapshot dependency will use the same snapshot of the sources. The bui
dependencies are built. If necessary, the dependencies will be triggered automatically.

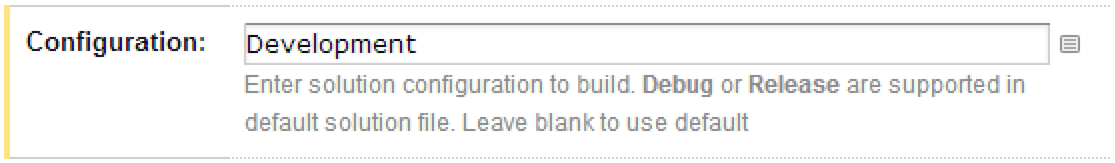
Edit Snapshot Dependency

Depend on: AcmeCorp.Portal :: Continuous Integration

Options:
- ✓ Do not run new build if there is a suitable one
  - ✓ Only use successful builds from suitable ones
- Run build even if dependency has failed
- Run build on the same agent

We want to be able to identify the build numbers throughout the entire chain of deployments. For example, if CI build 1.0.0 is deployed to staging, we want to be sure that this is actually version 1.0.0 and not some intermediate version. Under General Settings, change the build number format to use the same version number as the originating CI build. The build number format will have to be similar to %dep.WebAcmeCorpPortal_ContinuousIntegration.build.number%, duplicating the version number from the CI build.

Our CI build was building the default configuration for our solution. Since we are now deploying to a different environment and we've created deployment configurations (and configuration transforms) for Development and Production, let's change the build configuration through the Visual Studio build step.



Configuration: Development

Enter solution configuration to build. **Debug** or **Release** are supported in default solution file. Leave blank to use default

Now comes the actual deployment step! Up until now, we have built our project but we haven't really done anything to ship it to an actual server. Let's change that by adding a new build step based on a Command Line runner. As the build script, enter the following:

```
"C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe"
   -source:package='artifacts\WebDeploy\<target environment>\AcmeCompany.Portal.zip'
   -dest:auto,
      computerName="https://<windows azure web site web publish
URL>:443/msdeploy.axd?site=<windows azure web site name>",
      userName="<deployment user name>",
      password="<deployment password>",
      authtype="Basic",
      includeAcls="False"
   -verb:sync
   -disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension
   -setParamFile:"msdeploy\parameters\<target
environment>\AcmeCompany.Portal.SetParameters.xml"
```

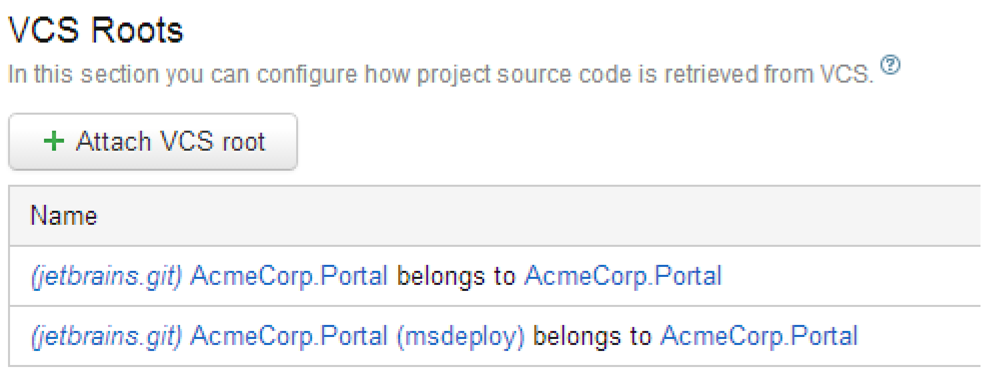
That's quite a bit, right? Let's go through this command:

- "C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe" is the path to the msdeploy.exe which has to be available on the build agent.
- -source:package='artifacts\WebDeploy\<target environment>\AcmeCompany.Portal.zip' specifies the deployment package we want to upload
- -dest:auto,computerName="https://<windows azure web site web publishURL>:443/msdeploy.axd?site=<windows azure web site name>",userName="<deployment user name>",password="<deployment password>",authtype="Basic",includeAcls="False" specifies the URL to the deployment service. For Windows Azure Web Sites, this will be in the aforementioned format. For IIS, this may be different (see Sayed Ibrahim Ashimi's excellent post on WebDeploy parameters)
- -verb:sync tells WebDeploy to synchronize only changed files (this will drastically reduce deployment time as not all files will be uploaded for every deployment)
- -disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension are used to disable certain configuration steps on the remote machine. These may be different for your environment, see MSDN for a complete list.
- -setParamFile:"msdeploy\parameters\<target environment>\AcmeCompany.Portal.SetParameters.xml" is an important one. It specifies the WebDeploy parameters that will be replaced in the deployed Web.config file on the remote server, for example the connection string. More on this file in a second.

The parameters file passed to the msdeploy.exe has to be created somehow. We've seen the build artifacts for our CI build contained a copy of this file and that one can be used if deployment secrets (such as the production database connection string) are available in source control. We probably don't want this, at least not in the same source control root our developers are all using.

Instead of storing passwords in a separate VCS root, they can also be added as a configuration parameter of type pass word in TeamCity. This will require creating the configuration file during the deployment, based on these configuration parameters.

For my setup, I've customized the AcmeCompany.Portal.SetParameters.xml file and put the configurations for the different target environments in a second VCS root, only available to the TeamCity server. This keeps the database connections strings a secret to everyone but TeamCity.

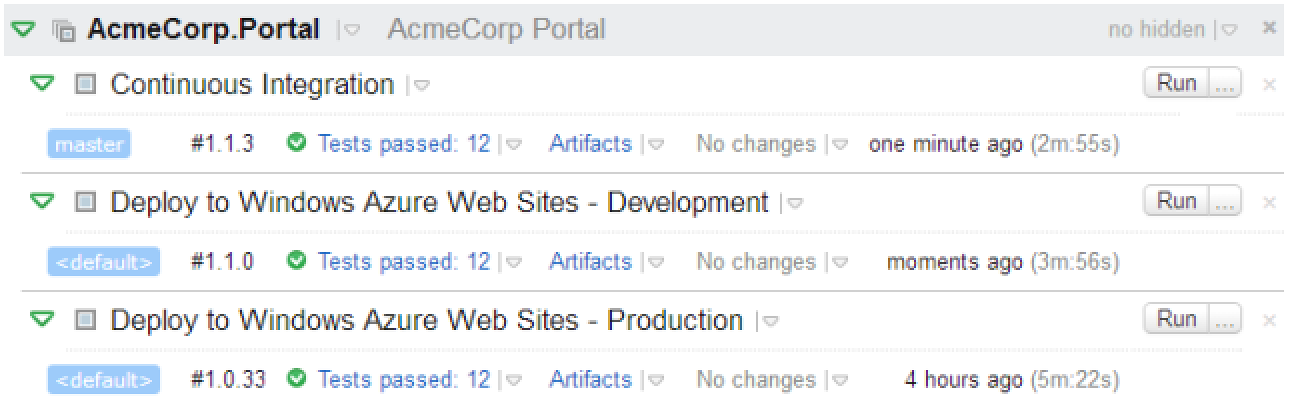

VCS Roots

In this section you can configure how project source code is retrieved from VCS.

Name
(jetbrains.git) AcmeCorp.Portal belongs to AcmeCorp.Portal
(jetbrains.git) AcmeCorp.Portal (msdeploy) belongs to AcmeCorp.Portal

We can repeat these steps to create a build configuration for staging, for QA, for production and so on. Since we want to promote builds over this entire chain, these configurations should all have a snapshot dependency on the previous environment.

Here's what this could look like: 3 different build configurations, denoting different versions that are deployed to each target environment:


AcmeCorp.Portal | AcmeCorp Portal — no hidden

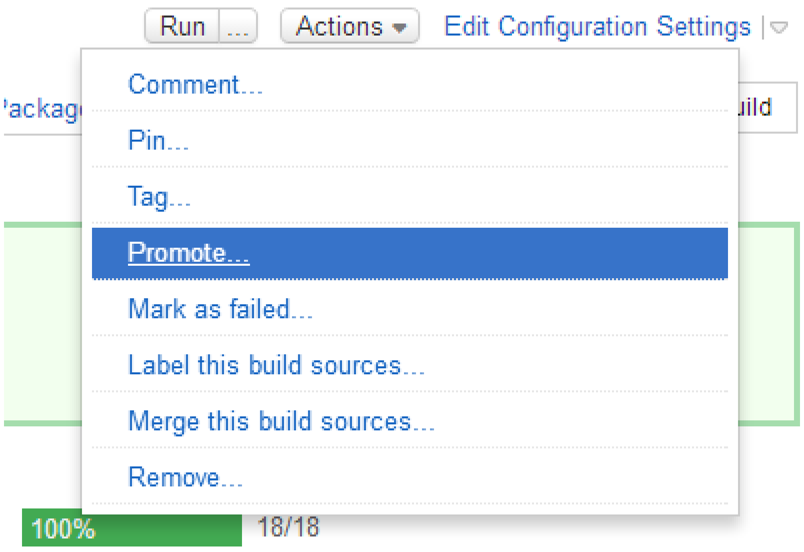
Continuous Integration
master #1.1.3 Tests passed: 12 | Artifacts | No changes | one minute ago (2m:55s)

Deploy to Windows Azure Web Sites - Development
<default> #1.1.0 Tests passed: 12 | Artifacts | No changes | moments ago (3m:56s)

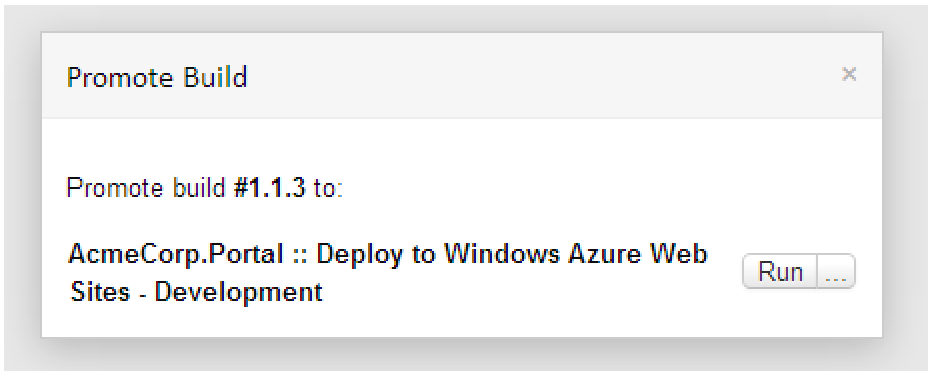
Deploy to Windows Azure Web Sites - Production
<default> #1.0.33 Tests passed: 12 | Artifacts | No changes | 4 hours ago (5m:22s)

Step 4: Promoting CI builds

Now that we have everything in place, let's see how we can promote builds from one environment to another. When we navigate to the build results of a CI build, we can use the Actions dropdown to promote our build to the next environment.


Run
...
Actions
Edit Configuration Settings
Comment...
Pin...
Tag...
Promote...
Mark as failed...
Label this build sources...
Merge this build sources...
Remove...
100%
18/18

Having configured the snapshot dependencies for our build configurations, TeamCity knows what the next environment should be: development.

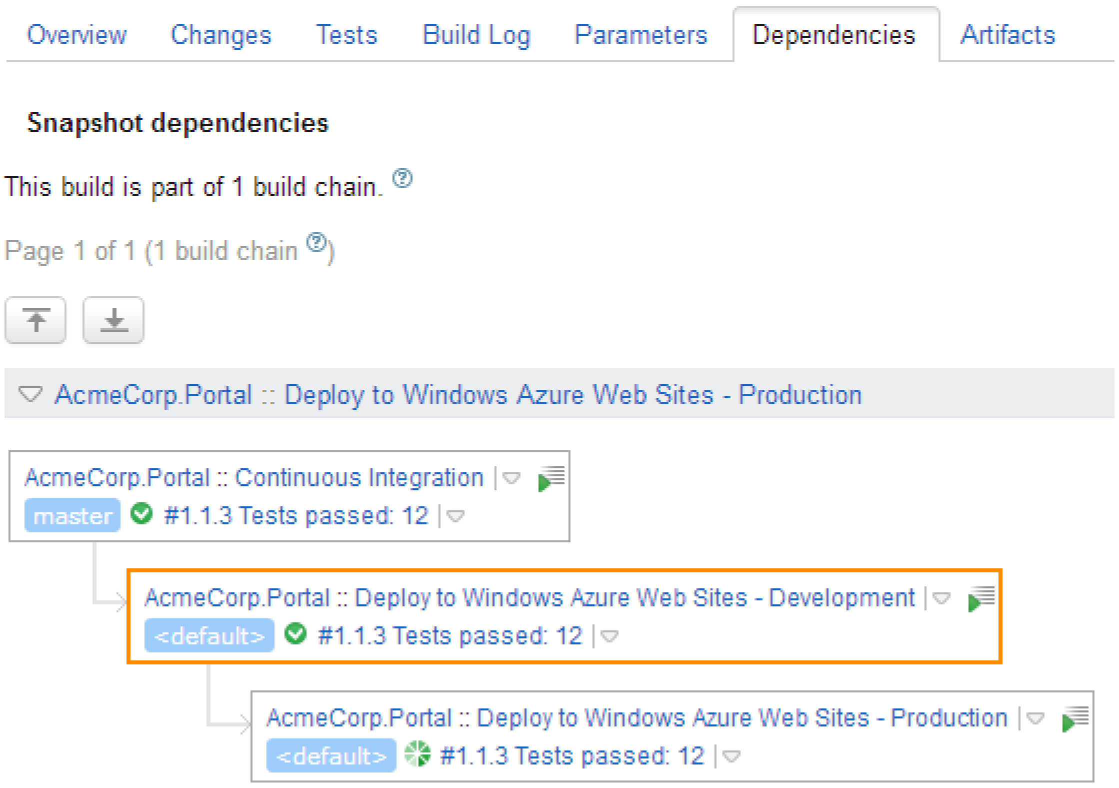

Promote Build

Promote build **#1.1.3** to:

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Development

This will trigger a new build that will deploy version 1.1.3 to the development environment. Once validated, we can navigate to that build's results and promote the build to the next environment.

Because of the snapshot dependencies we created, we can now also go to any build's Dependencies tab and see the environments where it has been deployed to. Here's build 1.1.3 as seen from development. We can see a CI build has been made, deployment to development has been done and deployment to production is still running:



Snapshot dependencies

This build is part of 1 build chain.

Page 1 of 1 (1 build chain)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Production

AcmeCorp.Portal :: Continuous Integration
master #1.1.3 Tests passed: 12

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Development
<default> #1.1.3 Tests passed: 12

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Production
<default> #1.1.3 Tests passed: 12

For a build configuration with snapshot dependencies, we can enable showing of changes from these dependencies using the Show changes from snapshot dependencies version control setting. This enables us to see exactly which changes are deployed. See Build Dependencies Setup - Changes from Dependencies for more information.

Conclusion

By thinking of a deployment as a chain of builds, doing deployments from TeamCity is not too hard. In this tutorial, we've used WebDeploy as an example means of transferring build artifacts to a target environment, but this could also have been another solution (like xcopy).

Using VCS labeling, it's also possible to label sources when a specific deployment happens. By pinning builds (optionally through the TeamCity API), we can make sure that build cleanup does not remove certain builds and artifacts.