

Setting up Episerver to Run as an Azure Application

By John Dymond | .NET Engineering Lead | Celerity | jdymond@celerity.com

Overview:

Azure applications offer an extremely scalable and inexpensive option for hosting Episerver web sites in the cloud. Compared with the cost of on-premise server and virtual machine options, as well as the evolution of Azure's application suite, it is a very compelling option for deploying your Episerver web sites.

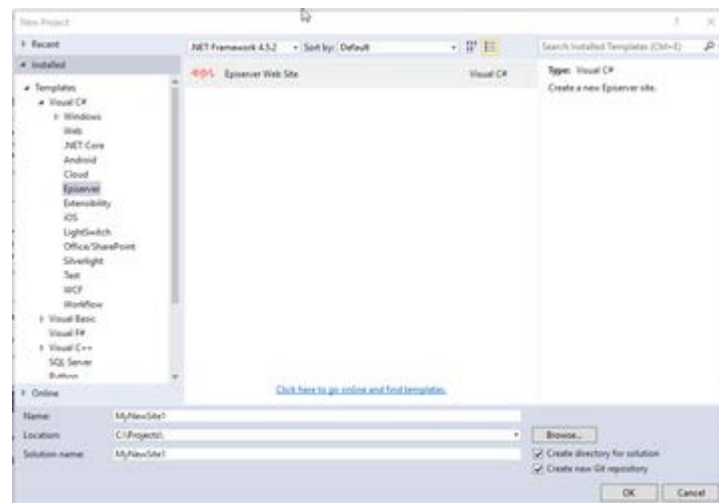
Below is a tutorial to do just that. This was inspired after realizing that the Episerver documentation for the process is a little outdated and the process has become more streamlined. Start with [Episerver's guide](#) as a refresher on the official documentation before proceeding.

The steps below assume that you have set up your Nuget package manager with the prerequisite packages - namely Episerver and Episerver Azure.

Steps:

1) Create your Episerver Project in Visual Studio

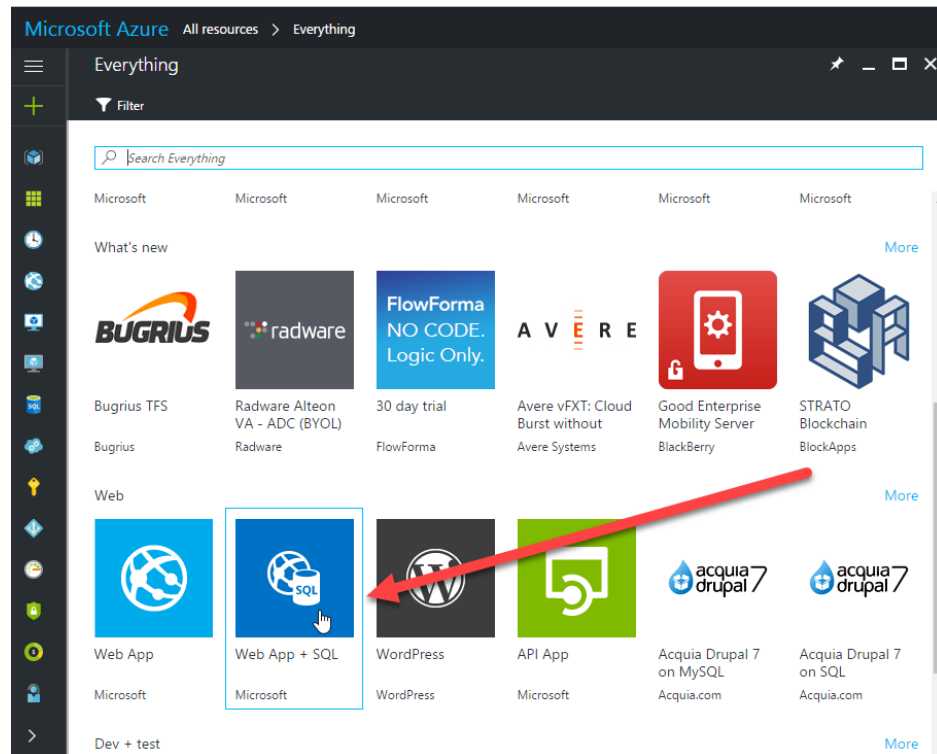
Start by creating an Episerver project in Visual Studio. I always check the box to create a new Git repository. This is a good practice whether or not you choose to use Continuous Integration later. It will save you from needing to set up source control manually later in the project.



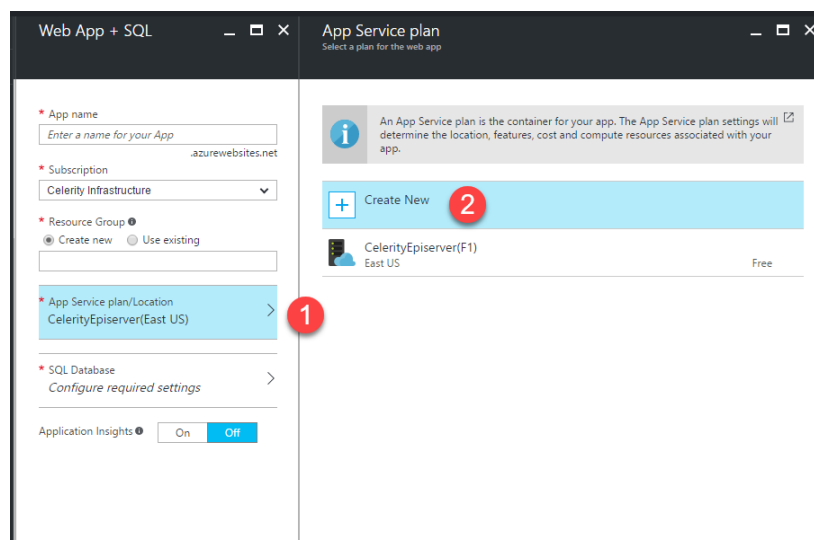
1. The documentation references creating an Alloy project, but I would recommend an empty project. It is easier to add pieces to an empty project than to remove them from Alloy. This option does not allow you to configure search at this time, but that is ok. You can set this up later.
2. Manually add your Episerver license file to the project. Add this to the root of the web application.

2) Set up Azure

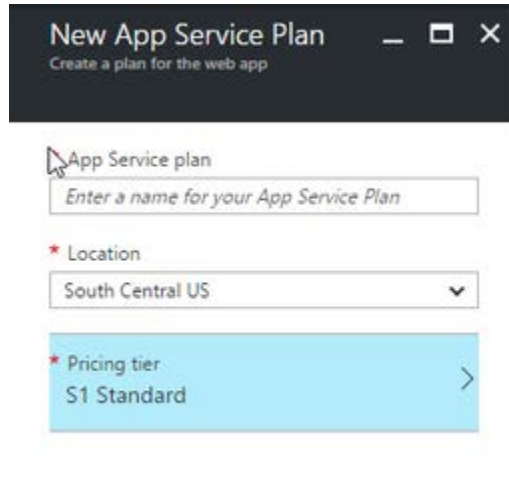
1. Log into the Azure portal. You now need to set up the environment where the application will reside. The application will be "Web App + SQL". You could independently set up the application and database, but this option streamlines the process and you will want both.



2. Assign a domain name and chose the proper subscription plan. Create a new plan or use an existing one.



3. Select your tier. This can be adjusted later, but it is worth taking some time to find the plan that makes the most sense for your expected site content and usage. Azure has a free tier for both Web Apps and Sql. This is good to use if you are just trying it out as a demo.



New App Service Plan
Create a plan for the web app

App Service plan
Enter a name for your App Service Plan

* Location
South Central US

* Pricing tier
S1 Standard

- Click the "View all" button to select the appropriate tier.

Choose your pricing tier

Browse the available plans and their features

App Service Environments are available in the Premium tier. They offer even greater scale options, private access, and more. [Learn more](#)

★ Recommended | [View all](#)

P1 Premium		P2 Premium		P3 Premium	
1	Core	2	Core	4	Core
1.75	GB RAM	3.5	GB RAM	7	GB RAM
BizTalk Services		BizTalk Services		BizTalk Services	
250 GB Storage		250 GB Storage		250 GB Storage	
Up to 20 instances * Subject to availability		Up to 20 instances * Subject to availability		Up to 20 instances * Subject to availability	
20 slots Web app staging		20 slots Web app staging		20 slots Web app staging	
50 times daily Backup		50 times daily Backup		50 times daily Backup	
Traffic Manager Geo availability		Traffic Manager Geo availability		Traffic Manager Geo availability	
223.20		446.40		892.80	
USD/MONTH (ESTIMATED)		USD/MONTH (ESTIMATED)		USD/MONTH (ESTIMATED)	

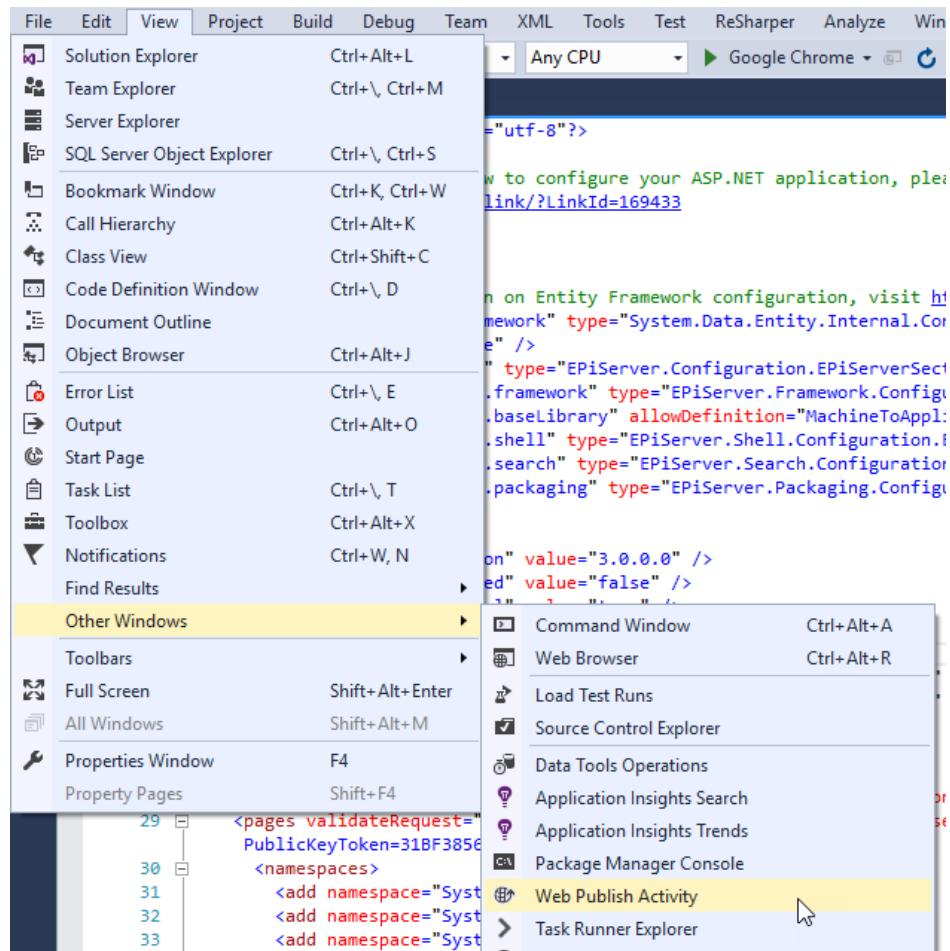
S1 Standard		S2 Standard		S3 Standard	
1	Core	2	Core	4	Core
1.75	GB RAM	3.5	GB RAM	7	GB RAM
50 GB Storage		50 GB Storage		50 GB Storage	
Custom domains / SSL SNI Incl & IP SSL Support		Custom domains / SSL SNI Incl & IP SSL Support		Custom domains / SSL SNI Incl & IP SSL Support	
Up to 10 instances Auto scale		Up to 10 instances Auto scale		Up to 10 instances Auto scale	
Daily Backup		Daily Backup		Daily Backup	
5 slots Web app staging		5 slots Web app staging		5 slots Web app staging	
Traffic Manager Geo availability		Traffic Manager Geo availability		Traffic Manager Geo availability	
74.40		148.80		297.60	
USD/MONTH (ESTIMATED)		USD/MONTH (ESTIMATED)		USD/MONTH (ESTIMATED)	

Select

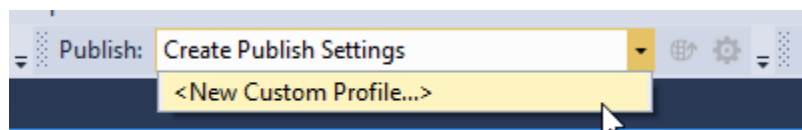
- Set up a database back on the main Web App + Sql frame. Follow the same steps as web site setup to establish a database. Databases also have a free tier, which is what I selected for this demonstration. Once you do this step you now have a running environment to deploy your Episerver project into the cloud. Now, we just need to hook together a few remaining pieces.

3) Establish a Publishing Profile

Back to Visual Studio: From here, you need to have web publishing enabled. If you do not, you can find it here:

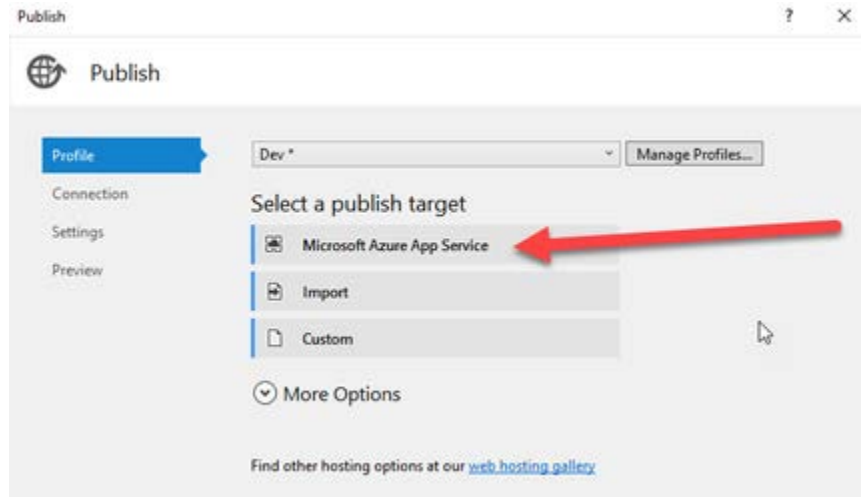


1. Click on Create Publish Settings. You will need to create a new profile if one does not already exist and just give it a name. Dev, Test, and Prod are common publish profiles.

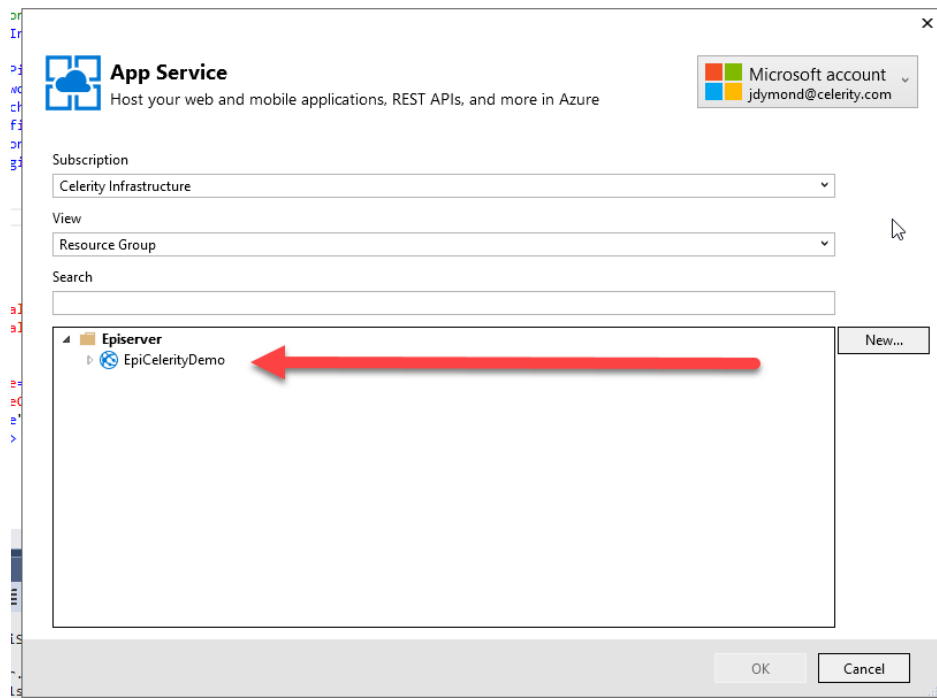


celerity

2. From here you can either choose to import a publish profile (from the Azure console) or use the App Service integration available. We are going to use the 2nd option and use the App Service.



3. To use this option, you will either need to login or be on the active account that has permission to access the Azure web app we set up previously. If done correctly, you should have no problems finding the application.



It is worth noting that the steps I presented are not the only way to set up your project with an Azure profile. It is worth exploring how to set up your Azure environment straight through the publish profile interface. The process is constantly improving and may be even more streamlined by the time you read this article.

4. When you browse the connection tab within the deployment wizard you should see all of your details auto-loaded. Proceed to settings.
5. Within settings, you should choose the configuration type you want for this publish profile. Because I am using this as a manner to publish up to the cloud (and do not intend to debug) I will choose the built in release configuration.
6. Refer to the Episerver documentation referenced at the top of this tutorial for database deployment setup. In this part we are defining how the database will be installed on the remote Azure environment. The only truly noteworthy thing of mention for these steps is that the 1st deployment is different than subsequent deployments. Once Episerver is up and running in the cloud, you will want to uncheck the Update database checkbox as we will not be making database changes on any subsequent deployments (unless attempting to upgrade).
7. In the preview tab of the publishing wizard we should be able to successfully preview both the file and database schema setup. Click publish and watch your site get deployed to Azure!

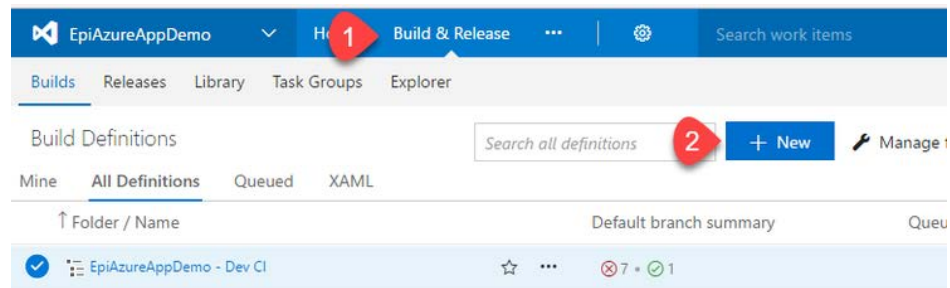
4) Continuous Integration

Continuous Integration (CI) is extremely useful in any project that has multiple team members. It means that deployments can be triggered by source control changes and avoids the possibility that multiple users try to deploy files at the same time (as it queues deployments). The only truly negative downside is that it is not always easy to automate every build process necessary to take code from source to build state and forces you to use a 100% automated approach to deployment vs. the inclusion of some manual steps. But the pain is well worth the gain as you will have an extremely streamlined deployment process and all settings within source control for retention purposes.

To set up CI for an Azure application, I am a big fan of using the Build & Release portions of TFS. These features can get extremely granular and support a ton of different applications that could be part of your build processes outside of .NET.

*Regardless of how you set up CI, you will need the admin configuring the builds to be an Azure "Global Admin." Otherwise, you will run into permissions issues and be unable to deploy your build. Refer to [this article](#) for more details.

1. Go to your project within Team Foundation Server. This tutorial is for the cloud version of TFS. If using the on-premise edition of TFS, the steps may vary greatly.
2. Go to the Build & Release tab and create a new definition.



3. There are many different definitions within the build definition template options. By default, Visual Studio is selected. This is the one we want. Click Next.
4. Choose the repository source, repository, branch, and check the box for Continuous Integration. Leave everything else as is, and click Create.

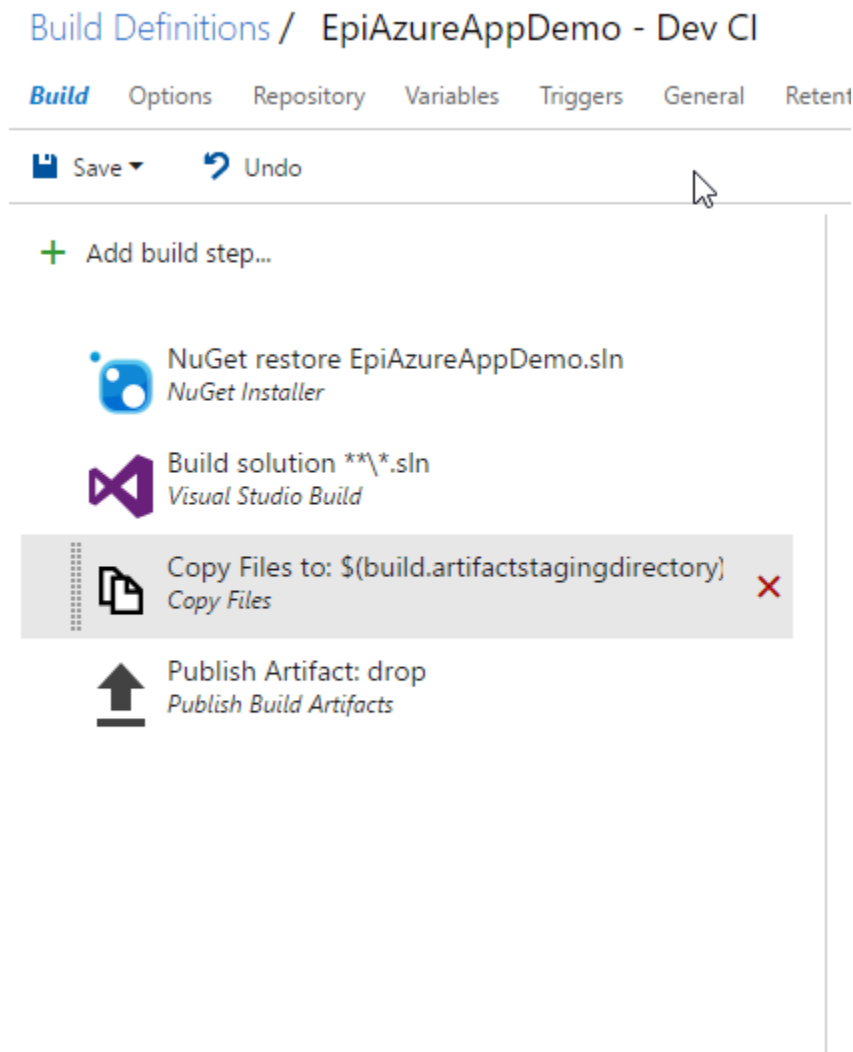
The screenshot shows the 'Create new build definition' dialog box. The 'Settings' section is expanded, showing the 'Repository source' options. The 'EpiAzureAppDemo Team Project' option is selected, indicated by a red circle with the number 1. Below this, the 'Repository' dropdown is set to 'EpiAzureAppDemo', highlighted with a red circle and the number 2. The 'Default branch' dropdown is set to 'master', highlighted with a red circle and the number 3. The 'Continuous integration (build whenever this branch is updated)' checkbox is checked, highlighted with a red circle and the number 4. The 'Default agent queue' is set to 'Hosted'. The 'Select folder' dropdown is set to '\'. At the bottom, there are buttons for '< Previous', 'Create', and 'Cancel'.

To recap what we just did: we created a brand new build definition, wired it to a branch

celerity

within source control, and told TFS to perform a build anytime a branch has accepted a code push. We are almost done now.

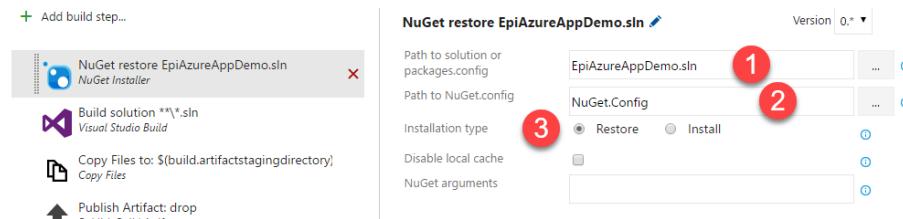
5. By default, TFS will configure a set of steps to perform on the build. Not all of these are needed and some will need tweaks in order to complete an Episerver deployment. Until you are comfortable with build definitions, I would recommend disabling build steps instead of deleting them. This will allow you to add them back quickly should you find you made a mistake while experimenting. For the purposes of this tutorial, you can delete the Test and Publish Symbols Path steps. These are not used in our example. Your build definition should now look something like this:



6. We are going to make some changes to the NuGet build step. This is not required for a deployment if you also manually deploy a project within TFS (outlined in the publish steps earlier). However, I feel it is important to self-contain your projects to be able to build everything through source control from the ground up. This is useful should you need to scale out at a later date.

Select a path to your project solution as well as a path to the NuGet.Config file. You may not be able to find a NuGet.Config file in your project or solution. I actually had to refer to the following article to be able to find the machine-wide NuGet.Config file and manually copy it to the solution folder. Nuget can be configured many different ways and is not in the scope of this tutorial. I chose the default setup, which puts the packages at the solution level (out of source control). Therefore, this build process chooses the Restore option for installation type.

Nuget.config location: <http://stackoverflow.com/questions/12836634/how-to-make-nuget-nuget-config-file-attached-to-vs-solution-to-be-not-ignored>



This step will likely be where you could run into issues and need tweaking as it is the only part of the project that involves customization to the point. You may need to make adjustments if that is the case.

The only thing I would mention under the build step is that you should verify that the Visual Studio version matches the version of Visual Studio you have. We need to ensure the exact same build engines are used or you could run into issues.

7. Save your build definition. It should automatically kick off a build. Queue a new build if it has not. If everything was configured properly you should see a green success status at the end of the build. If not, pay attention to the part of the build that failed and make adjustments to fix that step. Continue until you have reached a 100% success status. Once finished, this build process has been completed for the code check-in of every developer on the project! They will thank you later.
8. Congrats! If you have enabled the build use CI and configured it properly, you should now see the site updated after every successful build.

Important Factors to Consider

In a real world setting, you will want to have several build processes and several branches piloting those deployments. For releases in particular, you do not want to have such a streamlined approach from code check in to deployment. Instead, I am a much bigger fan of a production deployment from within Visual Studio itself or through planned releases. This manual process is far less likely to be done on accident and allows you to limit who has the ability to fire off a deployment of this sort.

We did not discuss transforms in this demo, but they play a significant part in this process as well. A typical solution will use CI in the dev and potentially test environments. In an Azure



application setting, this will likely mean different applications and settings for each environment. This can all be managed through transforms and publish settings. A side effect to this level of thoroughness is that all environmental settings will be source-controlled.

Lastly, application settings should be handled within the web.config and/or transform files whenever possible. Setting them up on the server can shield important details from other developers and does not lend itself well to replication should an application need to be torn down or stood up quickly. It is good practice that as many details as possible be available to source control and build processes.