CI/CD Pipelines for Team Foundation Services and .NET on Pivotal Cloud Foundry

Configuration Guide

Prepared by

Cornelius Mendoza
Sr. Platform Architect
cmendoza@pivotal.io

David Wu
Advisory Solutions Architect
dawu@pivotal.io

Introduction

Team Foundation Server (TFS) is a popular Source Code and Content Management system for companies developing applications with Microsoft technologies. Targeting Pivotal Cloud Foundry (PCF) installations from TFS can be easily accomplished using existing platform capabilities and available marketplace extensions. This guide is intended for an organization already using TFS and wants to establish a Continuous Integration and Continuous Delivery (CI/CD) pipeline for .NET and .NET core applications deploying into PCF installations.

Requirements

The following are required to successfully complete the configuration of the CI/CD pipeline in Team Foundation Services instructions found in this document. The steps have been successfully tested in both the public and private installation scenarios.

- Team Foundation Services 2017 SP1
    - Required for building .NET Core applications
    - A standalone server installation can be used in conjunction with other source code management systems or older versions of Team Foundation Services
- Cloud Foundry TFS Extension (available from marketplace)
- Visual Studio 2017 for build agents
- Cloud Foundry CLI for build agents
- Cloud Foundry Account(s) for deployment

New Windows Build Agents:

1. Create build machines and install the build agents following Microsoft's instructions found here.
2. Install the Cloud Foundry CLI for Windows on the build machine found here**.
3. Install any additional tools on the build machine that is required to complete code builds.

Existing Windows Build Agents:

1. Install the Cloud Foundry CLI on the build machine found here**.

**NOTE: The Cloud Foundry CLI can be pulled down from a repository during the deployment phase as an alternative to installing the CLI on each build machine. One advantage of pulling the CLI from the repository for each deployment is that it helps ensure the correct version is always used during the deployment process.

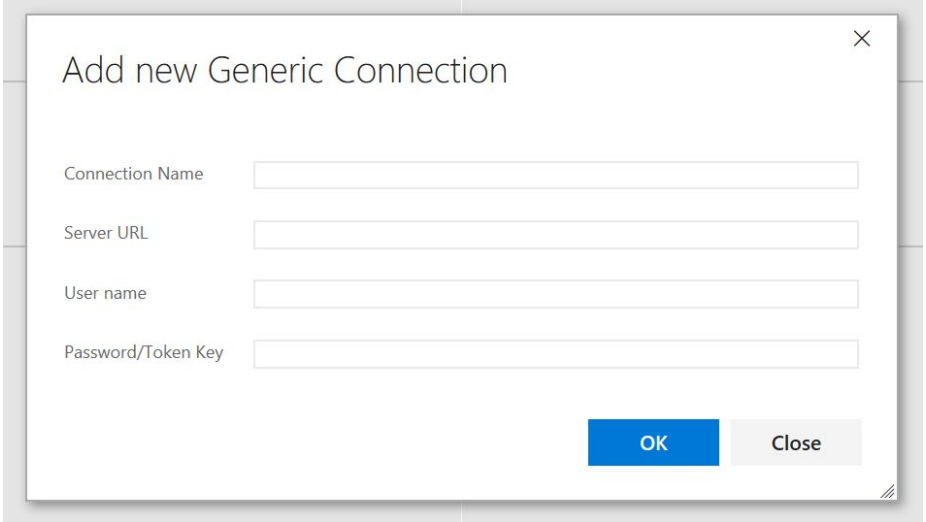
Server Endpoints:

To make PCF API endpoints available to TFS, the endpoints will need to be added to each project. To add the endpoints, login with using an account with permission to add Services.

To add a PCF Endpoint:

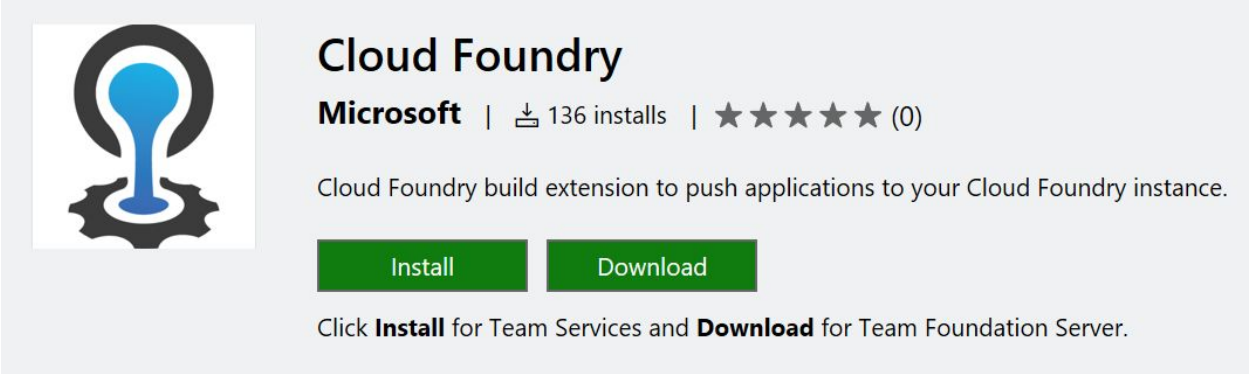
1. Select **+New Service Endpoint** in the Services page
2. Select **Generic** for the type

3. In the dialog box enter the information for the PCF API server endpoint
4. Select OK to save the information



Server Extensions:

To simplify the deployment using the Cloud Foundry CLI, the use of a Microsoft extension available from the marketplace is recommended. These steps can be accomplished through batch scripts but this approach would require extensive parameterization to support various PCF deployment scenarios. This Microsoft extension needs to be installed from the TFS Portal and can be found here.

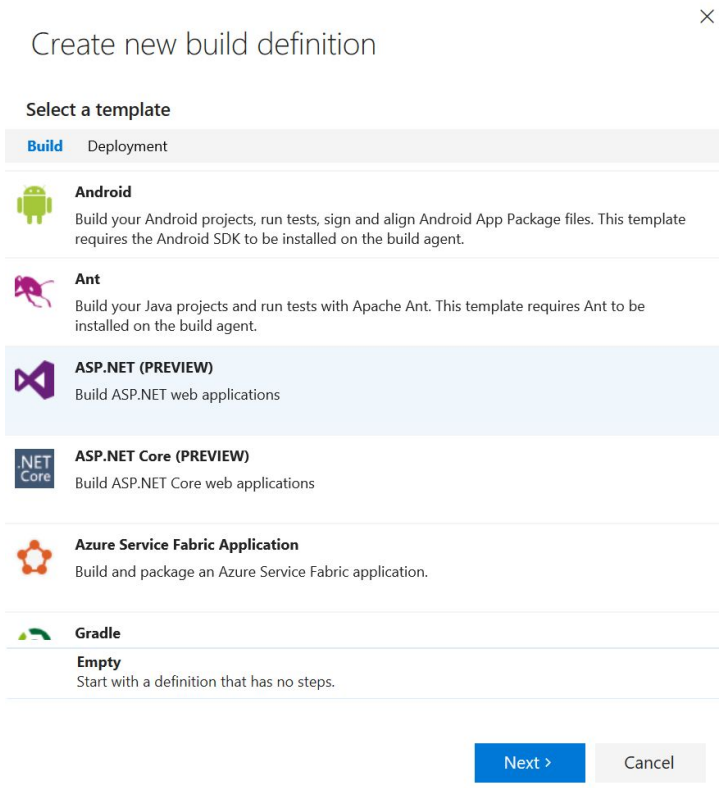


Building for Cloud Foundry

During the build process, changes will need to be made to the build definition. The goal of this step is to publish the application artifacts to a folder rather than the default of creating an IIS installation package.

.NET 4.52+ Application:

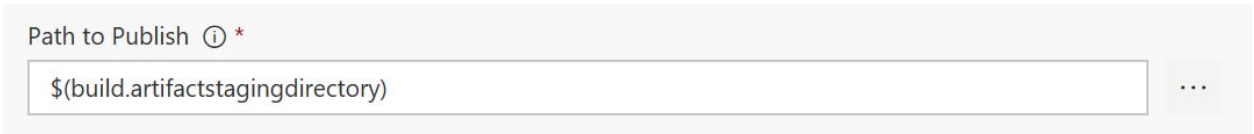
1. Create a new build definition and select ASP.NET Template



2. Attach project to the repository and select “Create”

3. Add the following arguments to the MSBuild task in the build definition. This will place the publish output into the artifact staging directory.
/p:DeployOnBuild=true /p:DeployDefaultTarget=WebPublish /p:WebPublishMethod=FileSystem /p:PackageAsSingleFile=false /p:SkipInvalidConfigurations=true /p:PublishUrl="$(build.artifactstagingdirectory)\\"

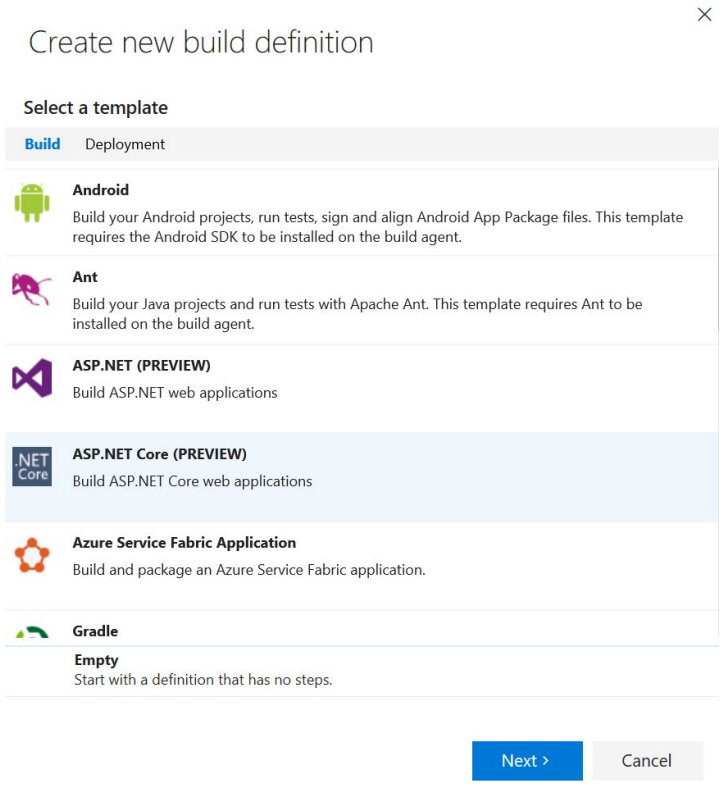
4. Add a Publish Artifact task if it is not already in the definition
5. If not already set, enter $(build.artifactstagingdirectory) in the Path to Publish field



6. Save the build definition

.NET Core Application:

1. Create a new build definition and select ASP.NET Core Template



2. Attach project to the repository and select “Create”
3. Modify the “Restore” task to use the .csproj project file instead of the project.json file

4. Modify the “Build” task to use the .csproj project file instead of the project.json file

5. Modify the “Test” task to use the .csproj project file instead of the “Test” project.json file

6. Modify the “Publish” task as follows:
    a. Uncheck the option for Publish Web Projects
    b. Uncheck the option for Zip Published Projects

7. Add a Publish Artifact task if it is missing from the definition and configure using step 5 from the .NET 4.52+ Application instructions
8. Save the build definition

Setting Up Releases

Pivotal Cloud Foundry deployments can be accomplished using the Cloud Foundry Extension. Users have a choice between deploying through a manifest file or through the configuration of the extension. To use a manifest file that is part of the project, include it with the published output of the project. .NET 4.x applications must run on Windows while .NET Core apps can run on Windows and Linux.

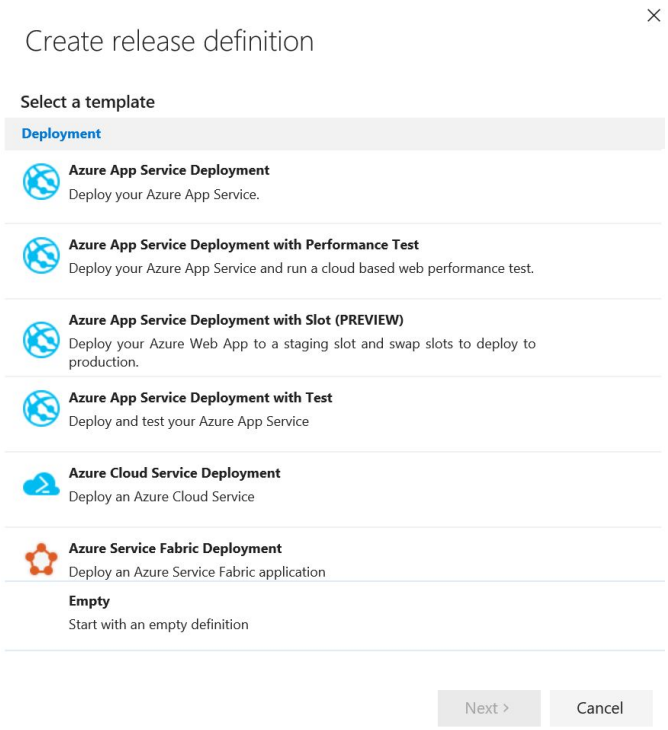
Example project file entry to include a manifest file:

```
<ItemGroup>
```

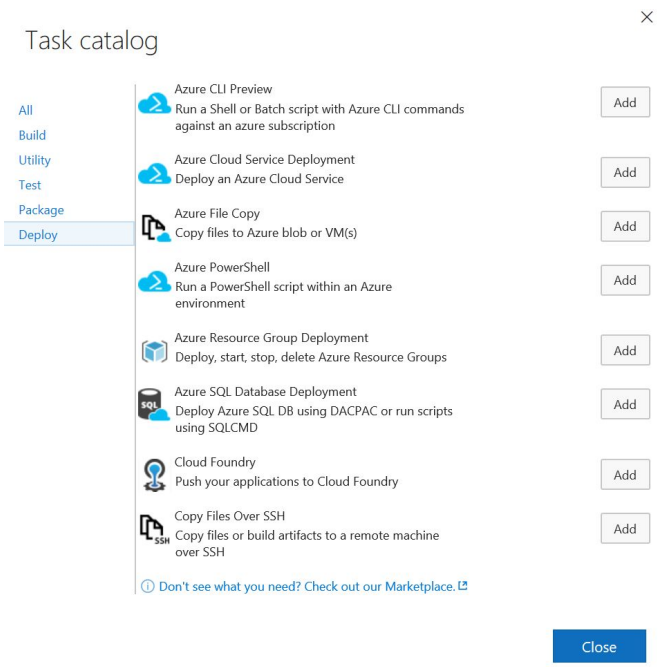
```
                <None Update="manifest.yml">
                <CopyToOutputDirectory>Always</CopyToOutputDirectory>
                </None>
    </ItemGroup>
```

.NET 4.52+ Application (No Manifest):

1. Create a new Release Definition for the project and select Empty template



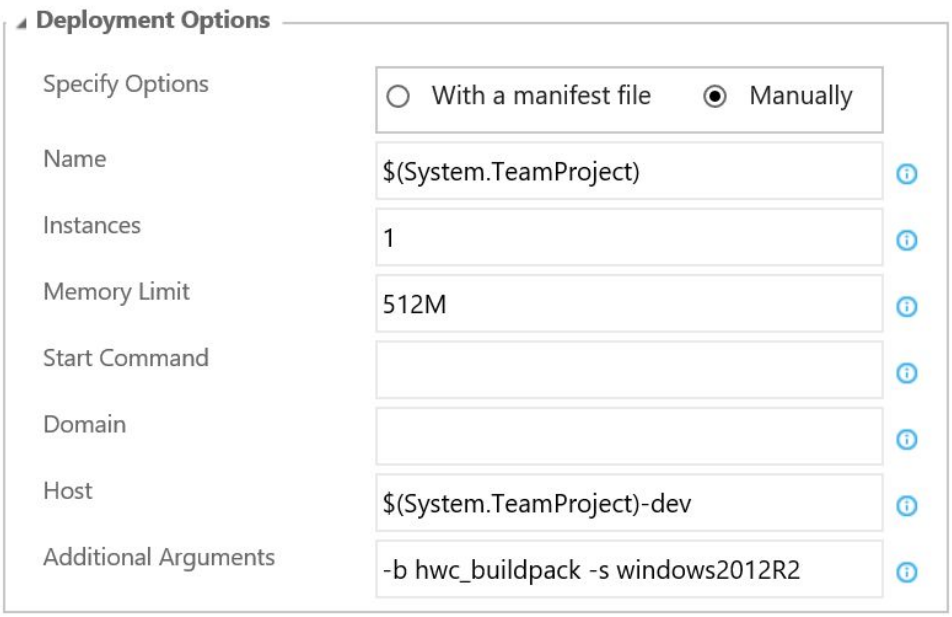
2. Select the project you wish to deploy and hit “Create”
3. Rename the default added environment “Development” – this environment is setup initially to be triggered whenever a new release is available
4. Select the “Run on Agent” phase and click “Add Task”
5. Find the Cloud Foundry push activity under *Deploy* and click “Add”



6. Select the endpoint for the deployment

7. Update connection settings to include the organization and space on the PCF installation you wish to deploy

8. Configure your deployment options to update the buildpack (e.g. -b hwc_buildpack) and target stack (e.g. -s windows2012R2) information for the application. Note 512Mb for memory limit should only be used as a guide only. Consider starting higher and scaling down to suit application.



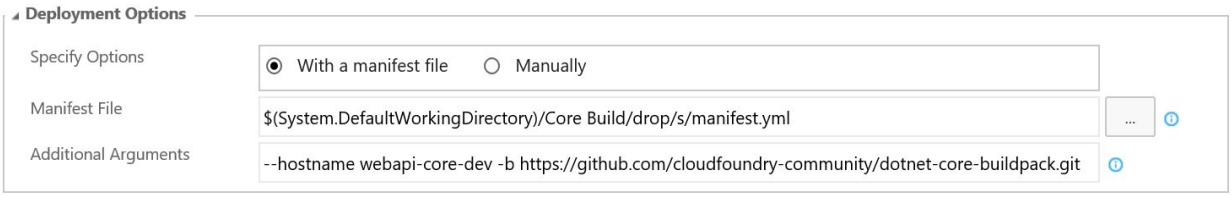
9. Set the working directory for the task to the “drop” folder created in the build – Tip: use the helper to locate the correct drop folder from the published artifacts

10. Add additional environments, updating the connection string and deployment options for each environment
11. Save the Release Definition

.NET Core Application (With Manifest):

The deployment of a .NET Core application is the same as deploying a .NET 4.x application with just one minor modification to the working folder. Configuring a deployment using a manifest will require adjustments to the deployment options of the extension.

1. Follow steps 1-7 as above
2. Configure your deployment options to specify the manifest file to use for the deployment and other arguments you wish to pass to the CLI – in this case, specifying a different hostname to use and a specific buildpack



3. Update the working directory to point to the directory with the source code published in the .NET Core build definition – Tip: use the helper to locate the correct source folder from the published artifacts

4. Save the Release Definition

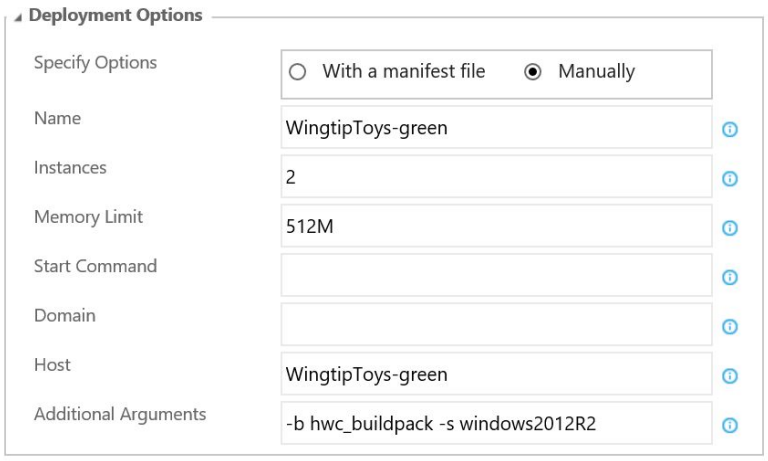
Configuring Blue/Green Deployments

TFS deployments can be created to allow for automated Blue/Green deployments in PCF. The following is a simple example of creating a Blue/Green CI/CD pipeline using TFS and the PCF extension to update routing after successful deployments. For more information on Blue/Green deployments, see the information located here.

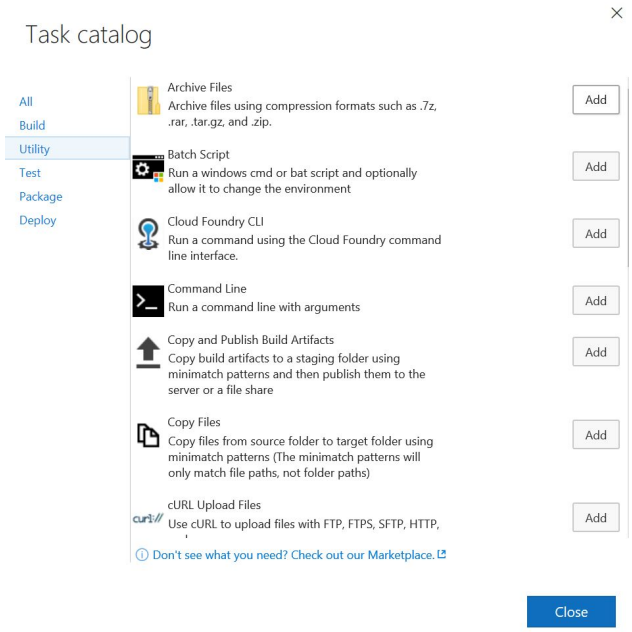
The following example is based on two production deployment instances called *WingtipToys-green* and *WingtipToys-blue*. The production hostname is *WingtipToys-prod* for this example.

Green Deployment

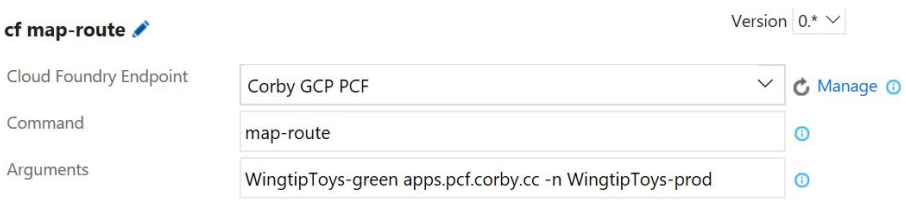
1. Create a new release definition and call it “Green Deploy”
2. Complete the build tasks from the Release section above based on the type of application you are deploying
3. In the *Deployment Options*, set the name and the hostname of the app to include “-green” to the end of the name



4. Add a CF tasks from the Utility tab to the deployment definition



5. Update the task command to be “map-route” and update the arguments to add the production route to the green deployment



6. The following can be done as part of the Green Deployment – The second phase in this

example was broken out to allow for an approval step

7. Add another environment to the Green Deployment and call it “Post Process”
8. Set the connection settings for the org and space in the connection options
9. Add a CF tasks from the Utility tab to the “Post Process” Run build agent and create a “map-route” task to re-register the blue route so that it can be accessible through a different hostname

10. Add another CF tasks for an “unmap-route” task to de-register the blue deployment from the production route

11. Add another CF tasks for an “unmap-route” task to de-register the green route

12. Save the deployment definition

Blue Deployment

1. Clone the “Green Deploy” release definition and call it “Blue Deploy”
2. Based on the steps above, replace the green definitions with the blue definitions and vice-versa in the task definitions
3. Save the deployment definition