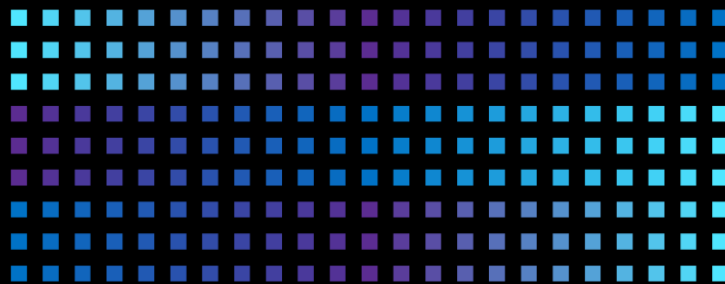




DevOps with ASP.NET Core and Azure



Cam Soper
Scott Addie

Microsoft Corporation

DevOps with ASP.NET Core and Azure

By [Cam Soper](#) and [Scott Addie](#)

Version 1.2.1

Updated September 7, 2018

The content in this book is open source. Review the license, view the latest updates, provide feedback, and propose changes at <https://aka.ms/aspnetdevops>.

PUBLISHED BY

Microsoft Developer Division, .NET, and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2018 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This book is provided "as-is" and expresses the author's views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at <https://www.microsoft.com> on the "Trademarks" webpage are trademarks of the Microsoft group of companies.

Mac and macOS are trademarks of Apple Inc.

All other marks and logos are property of their respective owners.

Acknowledgments

Thank you to everyone in the .NET community who contributed to this guide with helpful suggestions! We'd like to especially thank the following community members who contributed feedback on this material:

- [Sam Wronski](#)
- [Jeffrey Palermo](#)

Within Microsoft, the following people were instrumental for guidance, reviews, and encouragement:

- [Bill Wagner](#)
- [Cesar de la Torre](#)
- [Scott Hunter](#)
- [Wade Pickett](#)

Contents

Introduction	1
Who this guide is for	1
What this guide doesn't cover	1
What's in this guide	1
Tools and downloads	1
Deploy to App Service	1
Continuous integration and deployment.....	1
Monitor and debug	1
Next steps	1
Additional introductory reading	2
Tools and downloads	3
Prerequisites	3
Recommended tools (Windows only).....	3
Deploy an app to App Service	4
Download and test the app	4
Create the Azure App Service Web App	5
Deployment with Visual Studio.....	7
Deployment slots	10
Summary	13
Additional reading.....	14
Continuous integration and deployment.....	15
Publish the app's code to GitHub	15
Disconnect local Git deployment.....	16
Create a VSTS account	16
Configure the DevOps pipeline	17
Grant VSTS access to the GitHub repository.....	18
Create the build definition	19
Create the release pipeline	20
Commit changes to GitHub and automatically deploy to Azure	25
Examine the VSTS DevOps pipeline	27
Build definition.....	27
Release pipeline	30

Additional reading.....	33
Monitor and debug	34
Basic monitoring and troubleshooting	34
Advanced monitoring.....	36
Profile with Application Insights	36
Logging	41
Log streaming.....	41
Alerts	42
Live debugging	42
Conclusion.....	42
Additional reading.....	43
Next steps	44
Storage and databases.....	44
Identity.....	44
Mobile	44
Web infrastructure.....	44

Introduction

Welcome to the Azure Development Lifecycle guide for .NET! This guide introduces the basic concepts of building a development lifecycle around Azure using .NET tools and processes. After finishing this guide, you'll reap the benefits of a mature DevOps toolchain.

Who this guide is for

You should be an experienced ASP.NET developer (200-300 level). You don't need to know anything about Azure, as we'll cover that in this introduction. This guide may also be useful for DevOps engineers who are more focused on operations than development.

This guide targets Windows developers. However, Linux and macOS are fully supported by .NET Core. To adapt this guide for Linux/macOS, watch for callouts for Linux/macOS differences.

What this guide doesn't cover

This guide is focused on an end-to-end continuous deployment experience for .NET developers. It's not an exhaustive guide to all things Azure, and it doesn't focus extensively on .NET APIs for Azure services. The emphasis is all around continuous integration, deployment, monitoring, and debugging. Near the end of the guide, recommendations for next steps are offered. Included in the suggestions are Azure platform services that are useful to ASP.NET developers.

What's in this guide

Tools and downloads

Learn where to acquire the tools used in this guide.

Deploy to App Service

Learn the various methods for deploying an ASP.NET Core app to Azure App Service.

Continuous integration and deployment

Build an end-to-end continuous integration and deployment solution for your ASP.NET Core app with GitHub, VSTS, and Azure.

Monitor and debug

Use Azure's tools to monitor, troubleshoot, and tune your application.

Next steps

Other learning paths for the ASP.NET Core developer learning Azure.

Additional introductory reading

If this is your first exposure to cloud computing, these articles explain the basics.

- [What is Cloud Computing?](#)
- [Examples of Cloud Computing](#)
- [What is IaaS?](#)
- [What is PaaS?](#)

Tools and downloads

Azure has several interfaces for provisioning and managing resources, such as the [Azure portal](#), [Azure CLI](#), [Azure PowerShell](#), [Azure Cloud Shell](#), and Visual Studio. This guide takes a minimalist approach and uses the Azure Cloud Shell whenever possible to reduce the steps required. However, the Azure portal must be used for some portions.

Prerequisites

The following subscriptions are required:

- Azure — If you don't have an account, [get a free trial](#).
- Visual Studio Team Services (VSTS) — This account is created in Chapter 4.
- GitHub — If you don't have an account, [sign up for free](#).

The following tools are required:

- [Git](#) — A fundamental understanding of Git is recommended for this guide. Review the [Git documentation](#), specifically [git remote](#) and [git push](#).
- [.NET Core SDK](#) — Version 2.1.300 or later is required to build and run the sample app. If Visual Studio is installed with the **.NET Core cross-platform development** workload, the .NET Core SDK is already installed.

Verify your .NET Core SDK installation. Open a command shell, and run the following command:

```
dotnet --version
```

Recommended tools (Windows only)

- [Visual Studio](#)'s robust Azure tools provide a GUI for most of the functionality described in this guide. Any edition of Visual Studio will work, including the free Visual Studio Community Edition. The tutorials are written to demonstrate development, deployment, and DevOps both with and without Visual Studio.

Confirm that Visual Studio has the following [workloads](#) installed:

- ASP.NET and web development
- Azure development
- .NET Core cross-platform development

Deploy an app to App Service

[Azure App Service](#) is Azure's web hosting platform. Deploying a web app to Azure App Service can be done manually or by an automated process. This section of the guide discusses deployment methods that can be triggered manually or by script using the command line, or triggered manually using Visual Studio.

In this section, you'll accomplish the following tasks:

- Download and build the sample app.
- Create an Azure App Service Web App using the Azure Cloud Shell.
- Deploy the sample app to Azure using Git.
- Deploy a change to the app using Visual Studio.
- Add a staging slot to the web app.
- Deploy an update to the staging slot.
- Swap the staging and production slots.

Download and test the app

The app used in this guide is a pre-built ASP.NET Core app, [Simple Feed Reader](#). It's a Razor Pages app that uses the `Microsoft.SyndicationFeed.ReaderWriter` API to retrieve an RSS/Atom feed and display the news items in a list.

Feel free to review the code, but it's important to understand that there's nothing special about this app. It's just a simple ASP.NET Core app for illustrative purposes.

From a command shell, download the code, build the project, and run it as follows.

Note: Linux/macOS users should make appropriate changes for paths, e.g., using forward slash (/) rather than back slash (\).

1. Clone the code to a folder on your local machine.

```
git clone https://github.com/Azure-Samples/simple-feed-reader/
```

2. Change your working folder to the `simple-feed-reader` folder that was created.

```
cd .\simple-feed-reader\SimpleFeedReader
```

3. Restore the packages, and build the solution.

```
dotnet build
```

4. Run the app.

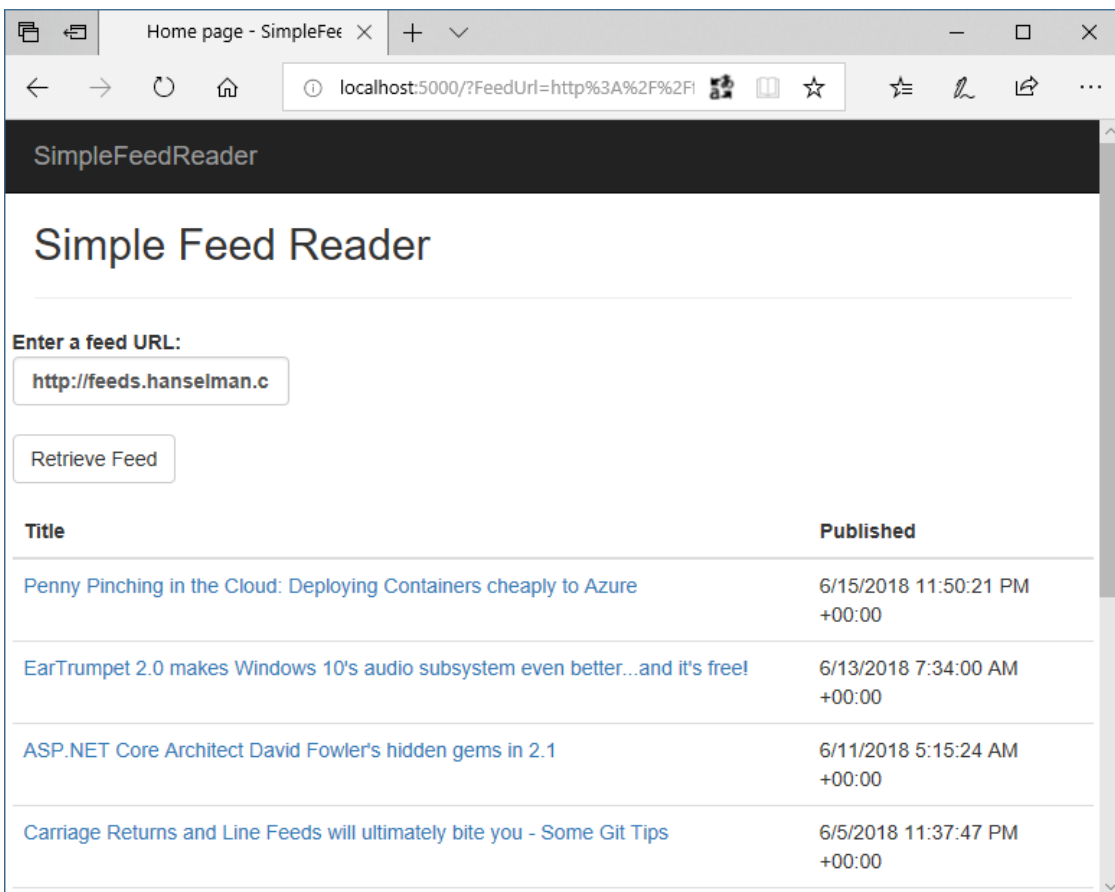
```
dotnet run
```

```
Command Prompt - dotnet run

C:\Src\simple-feed-reader\SimpleFeedReader>dotnet run
Hosting environment: Production
Content root path: C:\Src\simple-feed-reader\SimpleFeedReader
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

The dotnet run command is successful

5. Open a browser and navigate to `http://localhost:5000`. The app allows you to type or paste a syndication feed URL and view a list of news items.



The app displaying the contents of an RSS feed

6. Once you're satisfied the app is working correctly, shut it down by pressing **Ctrl+C** in the command shell.

Create the Azure App Service Web App

To deploy the app, you'll need to create an App Service [Web App](#). After creation of the Web App, you'll deploy to it from your local machine using Git.

1. Sign in to the [Azure Cloud Shell](#). Note: When you sign in for the first time, Cloud Shell prompts to create a storage account for configuration files. Accept the defaults or provide a unique name.
2. Use the Cloud Shell for the following steps.
 - a. Declare a variable to store your web app's name. The name must be unique to be used in the default URL. Using the `$RANDOM` Bash function to construct the name guarantees uniqueness and results in the format `webappname99999`.

```
webappname=mywebapp$RANDOM
```

- b. Create a resource group. Resource groups provide a means to aggregate Azure resources to be managed as a group.

```
az group create --location centralus --name AzureTutorial
```

The `az` command invokes the [Azure CLI](#). The CLI can be run locally, but using it in the Cloud Shell saves time and configuration.

- c. Create an App Service plan in the S1 tier. An App Service plan is a grouping of web apps that share the same pricing tier. The S1 tier isn't free, but it's required for the staging slots feature.

```
az appservice plan create --name $webappname --resource-group AzureTutorial --sku S1
```

- d. Create the web app resource using the App Service plan in the same resource group.

```
az webapp create --name $webappname --resource-group AzureTutorial --plan $webappname
```

- e. Set the deployment credentials. These deployment credentials apply to all the web apps in your subscription. Don't use special characters in the user name.

```
az webapp deployment user set --user-name REPLACE_WITH_USER_NAME --password REPLACE_WITH_PASSWORD
```

- f. Configure the web app to accept deployments from local Git and display the *Git deployment URL*. **Note this URL for reference later.**

```
echo Git deployment URL: $(az webapp deployment source config-local-git --name $webappname --resource-group AzureTutorial --query url --output tsv)
```

- g. Display the *web app URL*. Browse to this URL to see the blank web app. **Note this URL for reference later.**

```
echo Web app URL: http://$webappName.azurewebsites.net
```

3. Using a command shell on your local machine, navigate to the web app's project folder (for example, `.\simple-feed-reader\SimpleFeedReader`). Execute the following commands to set up Git to push to the deployment URL:

- a. Add the remote URL to the local repository.

```
git remote add azure-prod GIT_DEPLOYMENT_URL
```

- b. Push the local *master* branch to the *azure-prod* remote's *master* branch.

```
git push azure-prod master
```

You'll be prompted for the deployment credentials you created earlier. Observe the output in the command shell. Azure builds the ASP.NET Core app remotely.

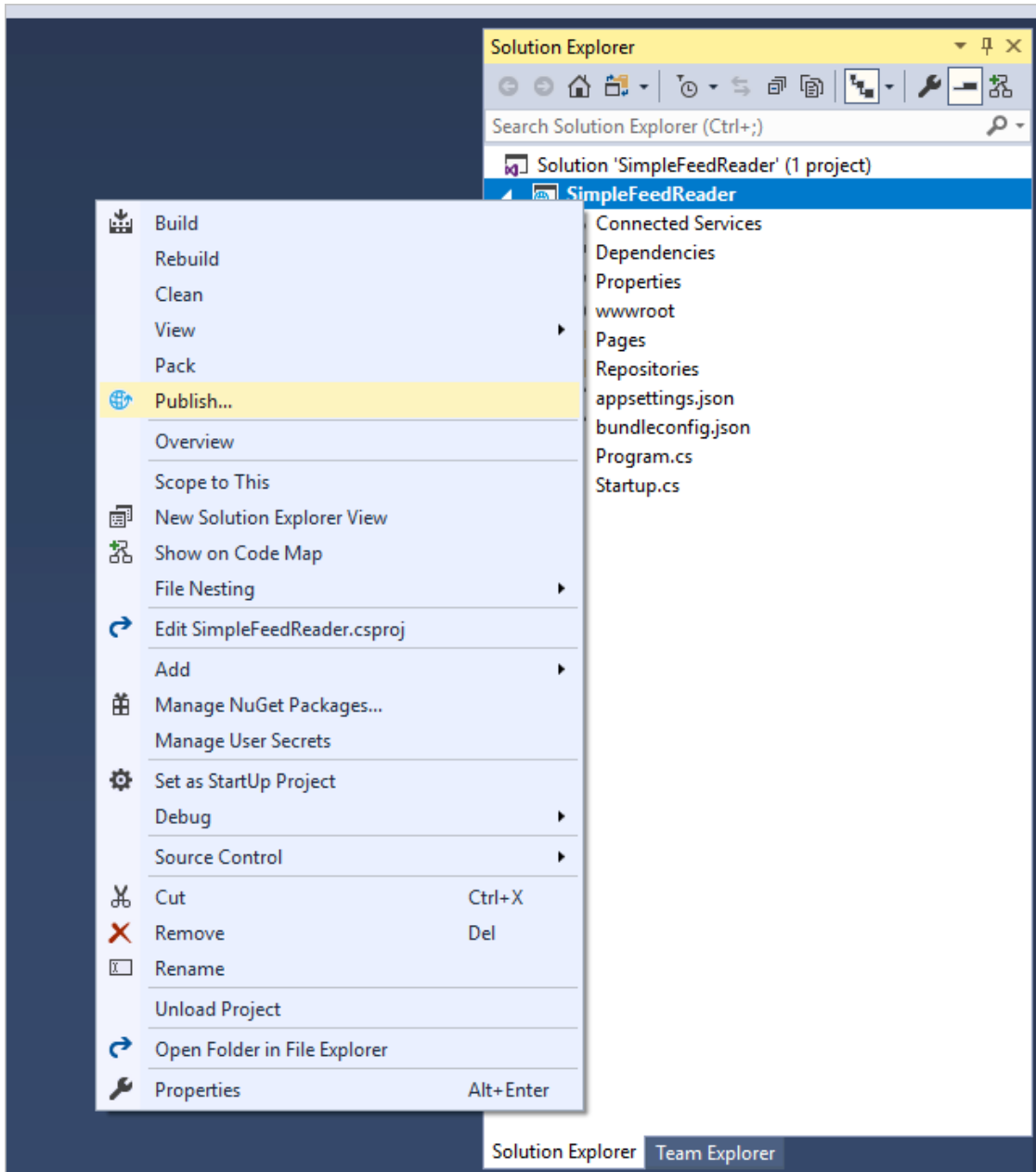
4. In a browser, navigate to the *Web app URL* and note the app has been built and deployed. Additional changes can be committed to the local Git repository with `git commit`. These changes are pushed to Azure with the preceding `git push` command.

Deployment with Visual Studio

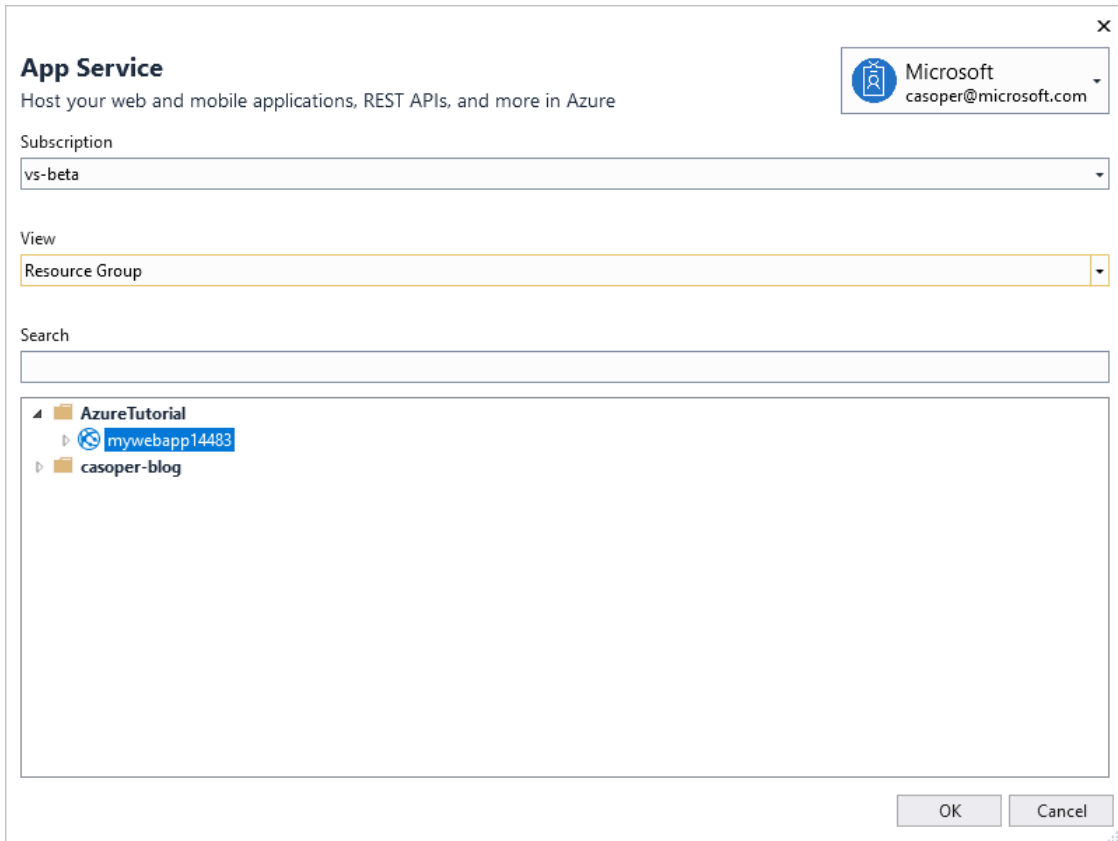
Note: This section applies to Windows only. Linux and macOS users should make the change described in step 2 below. Save the file, and commit the change to the local repository with `git commit`. Finally, push the change with `git push`, as in the first section.

The app has already been deployed from the command shell. Let's use Visual Studio's integrated tools to deploy an update to the app. Behind the scenes, Visual Studio accomplishes the same thing as the command line tooling, but within Visual Studio's familiar UI.

1. Open *SimpleFeedReader.sln* in Visual Studio.
2. In Solution Explorer, open *Pages.cshtml*. Change `<h2>Simple Feed Reader</h2>` to `<h2>Simple Feed Reader - V2</h2>`.
3. Press **Ctrl+Shift+B** to build the app.
4. In Solution Explorer, right-click on the project and click **Publish**.

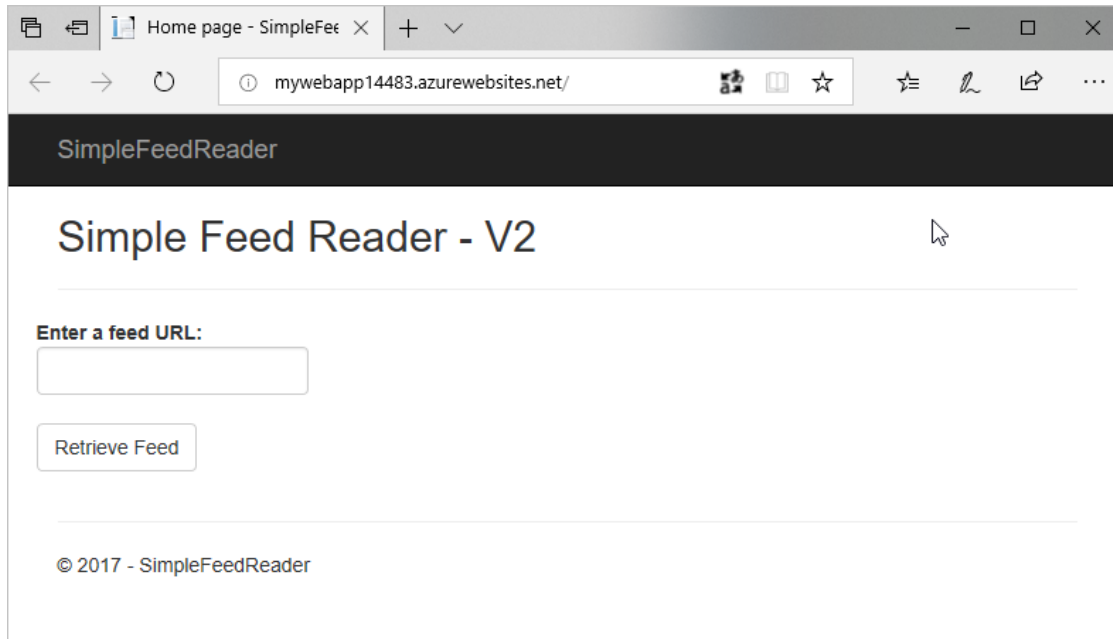


5. Visual Studio can create a new App Service resource, but this update will be published over the existing deployment. In the **Pick a publish target** dialog, select **App Service** from the list on the left, and then select **Select Existing**. Click **Publish**.
6. In the **App Service** dialog, confirm that the Microsoft or Organizational account used to create your Azure subscription is displayed in the upper right. If it's not, click the drop-down and add it.
7. Confirm that the correct Azure **Subscription** is selected. For **View**, select **Resource Group**. Expand the **AzureTutorial** resource group and then select the existing web app. Click **OK**.



Publish App Service dialog

Visual Studio builds and deploys the app to Azure. Browse to the web app URL. Validate that the <h2> element modification is live.



The app with the changed title

Deployment slots

Deployment slots support the staging of changes without impacting the app running in production. Once the staged version of the app is validated by a quality assurance team, the production and staging slots can be swapped. The app in staging is promoted to production in this manner. The following steps create a staging slot, deploy some changes to it, and swap the staging slot with production after verification.

1. Sign in to the [Azure Cloud Shell](#), if not already signed in.
2. Create the staging slot.
 - a. Create a deployment slot with the name *staging*.

```
az webapp deployment slot create --name $webappname --resource-group AzureTutorial --slot staging
```

- b. Configure the staging slot to use deployment from local Git and get the **staging** deployment URL. **Note this URL for reference later.**

```
echo Git deployment URL for staging: $(az webapp deployment source config -local-git --name $webappname --resource-group AzureTutorial --slot staging --query url --output tsv)
```

- c. Display the staging slot's URL. Browse to the URL to see the empty staging slot. **Note this URL for reference later.**

```
echo Staging web app URL: http://$webappname-staging.azurewebsites.net
```

3. In a text editor or Visual Studio, modify *Pages/Index.cshtml* again so that the `<h2>` element reads `<h2>Simple Feed Reader - V3</h2>` and save the file.

4. Commit the file to the local Git repository, using either the **Changes** page in Visual Studio's *Team Explorer* tab, or by entering the following using the local machine's command shell:

```
git commit -a -m "upgraded to V3"
```

5. Using the local machine's command shell, add the staging deployment URL as a Git remote and push the committed changes:

a. Add the remote URL for staging to the local Git repository.

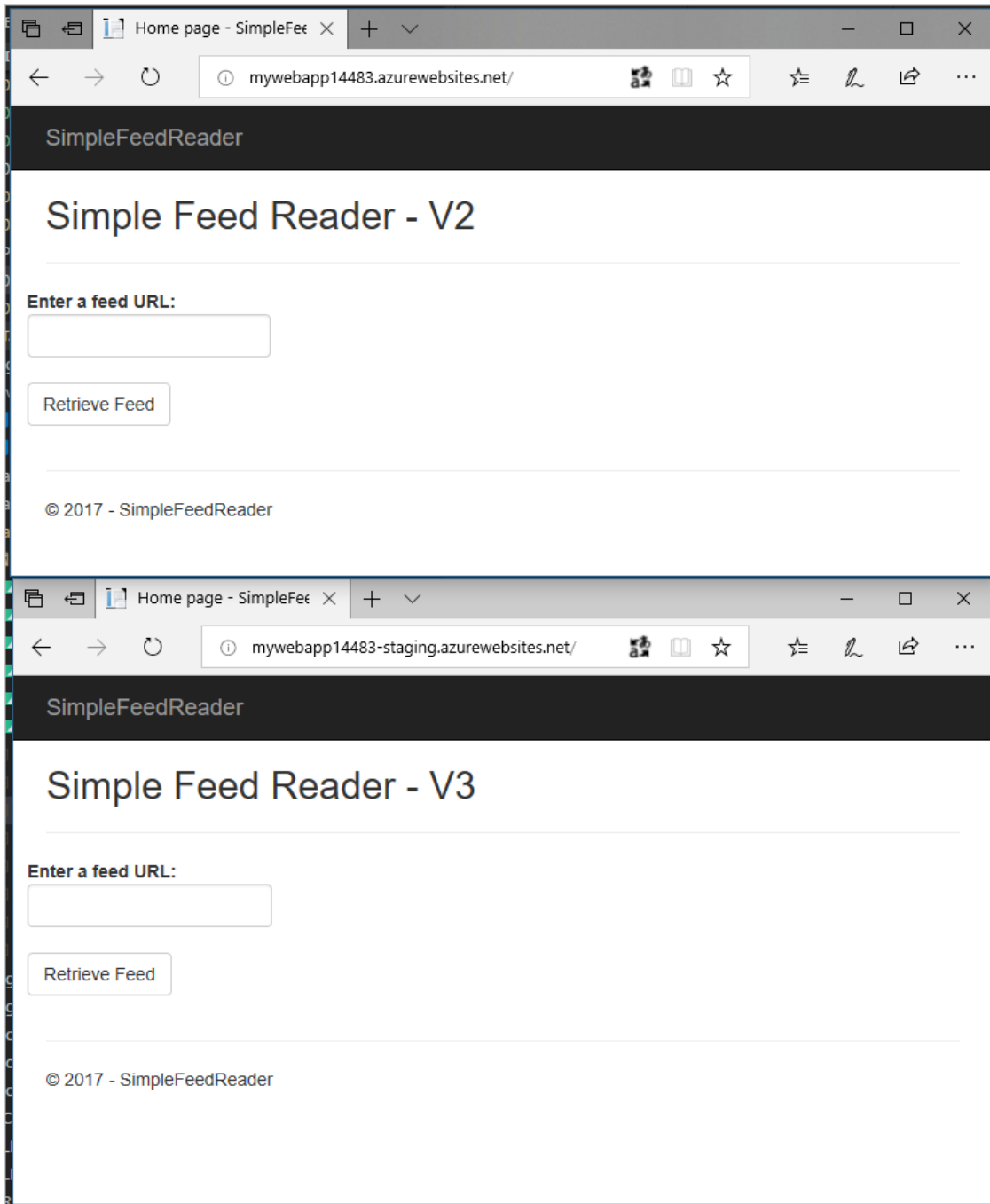
```
git remote add azure-staging <Git_staging_deployment_URL>
```

b. Push the local *master* branch to the *azure-staging* remote's *master* branch.

```
git push azure-staging master
```

Wait while Azure builds and deploys the app.

6. To verify that V3 has been deployed to the staging slot, open two browser windows. In one window, navigate to the original web app URL. In the other window, navigate to the staging web app URL. The production URL serves V2 of the app. The staging URL serves V3 of the app.

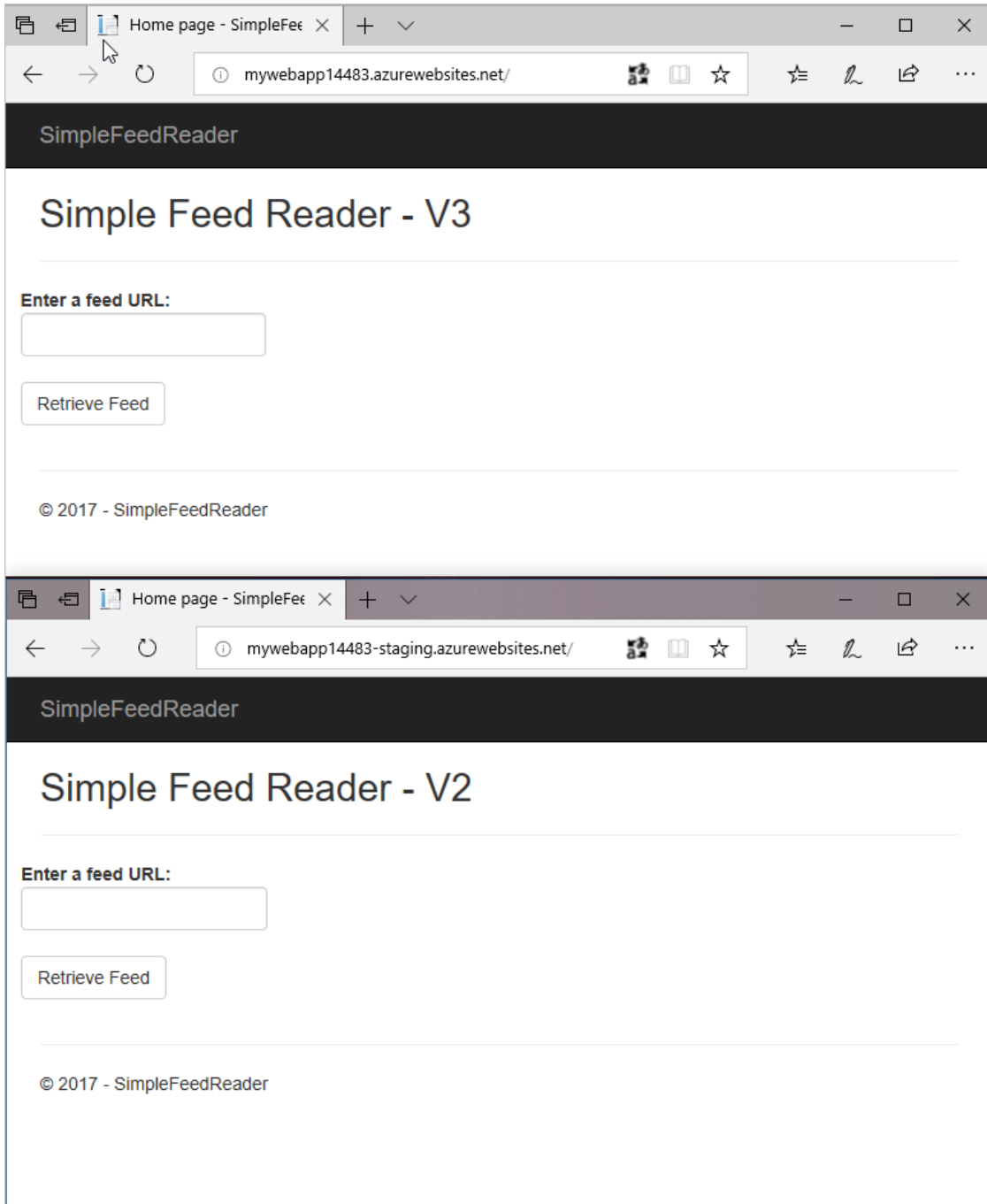


Comparing the browser windows

7. In the Cloud Shell, swap the verified/warmed-up staging slot into production.

```
az webapp deployment slot swap --name $webappname --resource-group AzureT  
utorial --slot staging
```

8. Verify that the swap occurred by refreshing the two browser windows.



Comparing the browser windows after the swap

Summary

In this section, the following tasks were completed:

- Downloaded and built the sample app.
- Created an Azure App Service Web App using the Azure Cloud Shell.
- Deployed the sample app to Azure using Git.

- Deployed a change to the app using Visual Studio.
- Added a staging slot to the web app.
- Deployed an update to the staging slot.
- Swapped the staging and production slots.

In the next section, you'll learn how to build a DevOps pipeline with Azure and Visual Studio Team Services.

Additional reading

- [Web Apps overview](#)
- [Build a .NET Core and SQL Database web app in Azure App Service](#)
- [Configure deployment credentials for Azure App Service](#)
- [Set up staging environments in Azure App Service](#)

Continuous integration and deployment

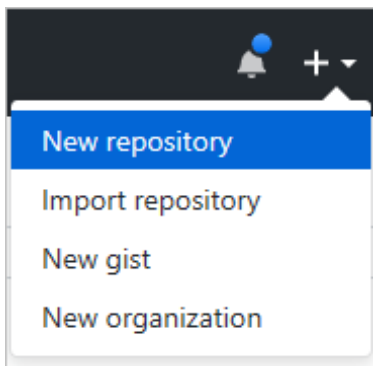
In the previous chapter, you created a local Git repository for the Simple Feed Reader app. In this chapter, you'll publish that code to a GitHub repository and construct a Visual Studio Team Services (VSTS) DevOps pipeline. The pipeline enables continuous builds and deployments of the app. Any commit to the GitHub repository triggers a build and a deployment to the Azure Web App's staging slot.

In this section, you'll complete the following tasks:

- Publish the app's code to GitHub
- Disconnect local Git deployment
- Create a VSTS account
- Create a team project in VSTS
- Create a build definition
- Create a release pipeline
- Commit changes to GitHub and automatically deploy to Azure
- Examine the VSTS DevOps pipeline

Publish the app's code to GitHub

9. Open a browser window, and navigate to <https://github.com>.
10. Click the + drop-down in the header, and select **New repository**:



GitHub New Repository option

11. Select your account in the **Owner** drop-down, and enter *simple-feed-reader* in the **Repository name** textbox.
12. Click the **Create repository** button.
13. Open your local machine's command shell. Navigate to the directory in which the *simple-feed-reader* Git repository is stored.
14. Rename the existing *origin* remote to *upstream*. Execute the following command:

```
console git remote rename origin upstream
```

15. Add a new *origin* remote pointing to your copy of the repository on GitHub. Execute the following command: `console git remote add origin https://github.com/<GitHub_username>/simple-feed-reader/`
16. Publish your local Git repository to the newly created GitHub repository. Execute the following command: `console git push -u origin master`
17. Open a browser window, and navigate to `https://github.com/<GitHub_username>/simple-feed-reader/`. Validate that your code appears in the GitHub repository.

Disconnect local Git deployment

Remove the local Git deployment with the following steps. VSTS both replaces and augments that functionality.

18. Open the [Azure portal](#), and navigate to the *staging (mywebapp<unique_number>/staging)* Web App. The Web App can be quickly located by entering *staging* in the portal's search box:



staging Web App search term

19. Click **Deployment options**. A new panel appears. Click **Disconnect** to remove the local Git source control configuration that was added in the previous chapter. Confirm the removal operation by clicking the **Yes** button.
20. Navigate to the *mywebapp* App Service. As a reminder, the portal's search box can be used to quickly locate the App Service.
21. Click **Deployment options**. A new panel appears. Click **Disconnect** to remove the local Git source control configuration that was added in the previous chapter. Confirm the removal operation by clicking the **Yes** button.

Create a VSTS account

22. Open a browser, and navigate to the [VSTS account creation page](#).
23. Type a unique name into the **Pick a memorable name** textbox to form the URL for accessing your VSTS account.
24. Select the **Git** radio button, since the code is hosted in a GitHub repository.
25. Click the **Continue** button. After a short wait, an account and a team project, named *MyFirstProject*, are created.

Host my projects at:

.visualstudio.com

Manage code using:

- Git
- Team Foundation Version Control

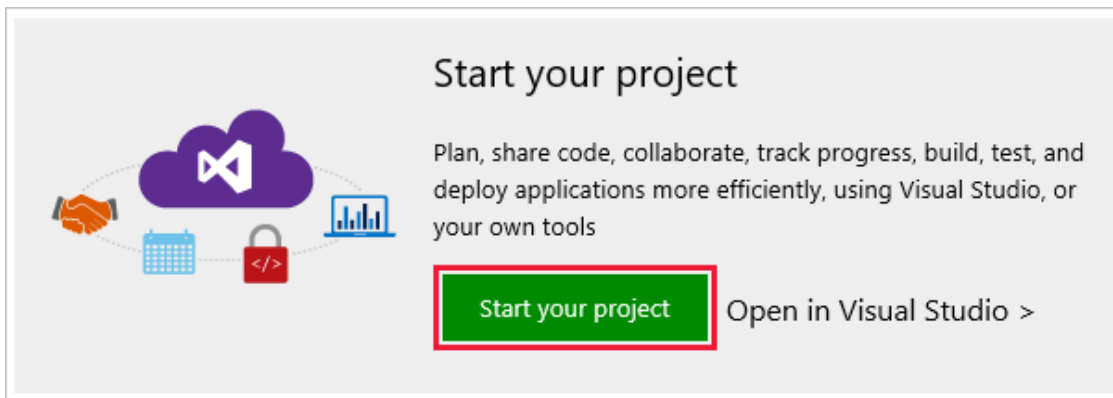
We will host your projects in **Central US** location.
 You can share work with other **Microsoft** users.

✎ Change details

Continue

VSTS account creation page

26. Open the confirmation email indicating that the VSTS account and project are ready for use. Click the **Start your project** button:



Start your project button

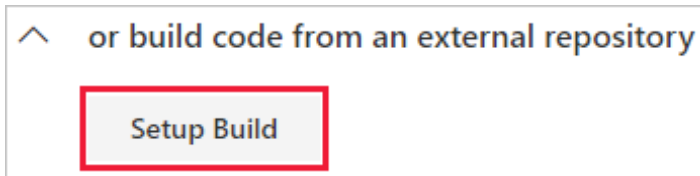
27. A browser opens to `<account_name>.visualstudio.com`. Click the *MyFirstProject* link to begin configuring the project's DevOps pipeline.

Configure the DevOps pipeline

There are three distinct steps to complete. Completing the steps in the following three sections results in an operational DevOps pipeline.

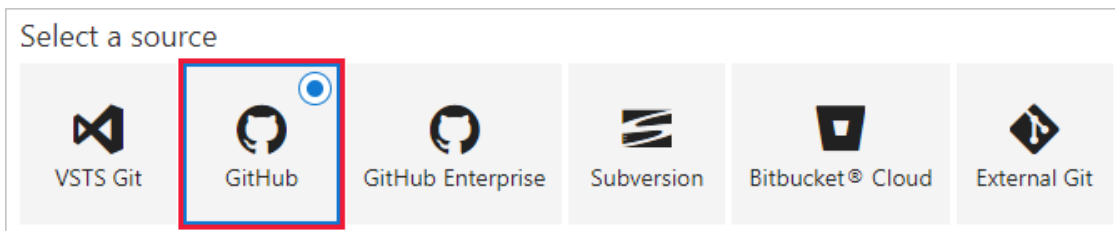
Grant VSTS access to the GitHub repository

- Expand the **or build code from an external repository** accordion. Click the **Setup Build** button:



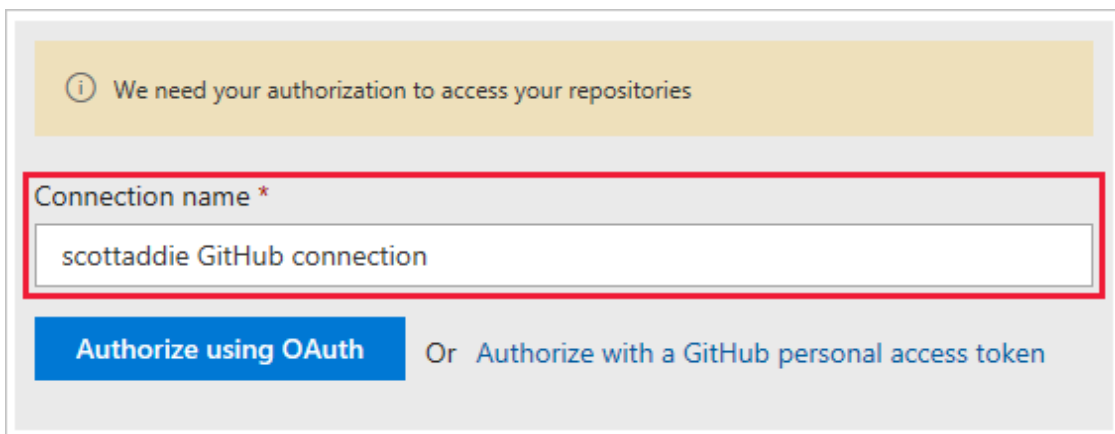
Setup Build button

- Select the **GitHub** option from the **Select a source** section:



Select a source - GitHub

- Authorization is required before VSTS can access your GitHub repository. Enter *GitHub connection* in the **Connection name** textbox. For example:



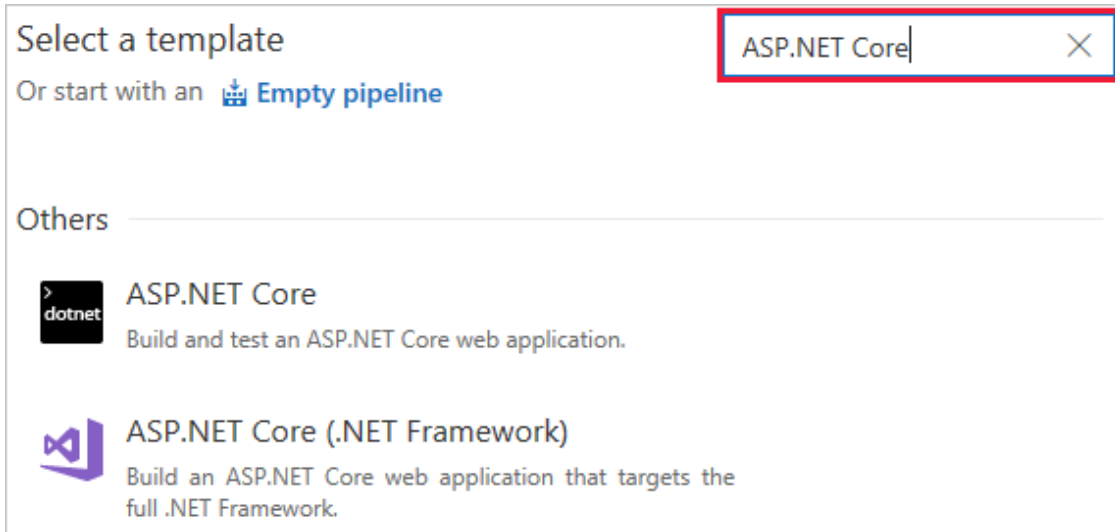
GitHub connection name

- If two-factor authentication is enabled on your GitHub account, a personal access token is required. In that case, click the **Authorize with a GitHub personal access token** link. See the [official GitHub personal access token creation instructions](#) for help. Only the *repo* scope of permissions is needed. Otherwise, click the **Authorize using OAuth** button.
- When prompted, sign in to your GitHub account. Then select Authorize to grant access to your VSTS account. If successful, a new service endpoint is created.
- Click the ellipsis button next to the **Repository** button. Select the */simple-feed-reader* repository from the list. Click the **Select** button.

34. Select the *master* branch from the **Default branch for manual and scheduled builds** drop-down. Click the **Continue** button. The template selection page appears.

Create the build definition

35. From the template selection page, enter *ASP.NET Core* in the search box:



Select a template

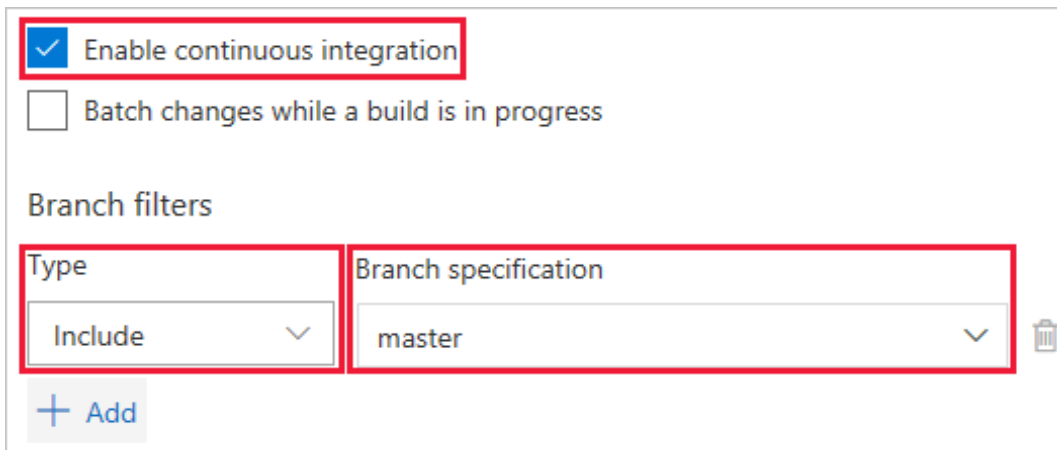
Or start with an **Empty pipeline**

Others

- ASP.NET Core**
Build and test an ASP.NET Core web application.
- ASP.NET Core (.NET Framework)**
Build an ASP.NET Core web application that targets the full .NET Framework.

ASP.NET Core search on template page

36. The template search results appear. Hover over the **ASP.NET Core** template, and click the **Apply** button.
37. The **Tasks** tab of the build definition appears. Click the **Triggers** tab.
38. Check the **Enable continuous integration** box. Under the **Branch filters** section, confirm that the **Type** drop-down is set to *Include*. Set the **Branch specification** drop-down to *master*.



Enable continuous integration

Batch changes while a build is in progress

Branch filters

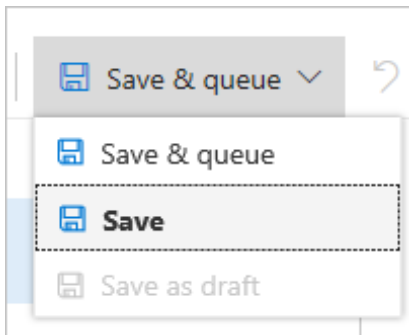
Type	Branch specification
Include	master

+ Add

Enable continuous integration settings

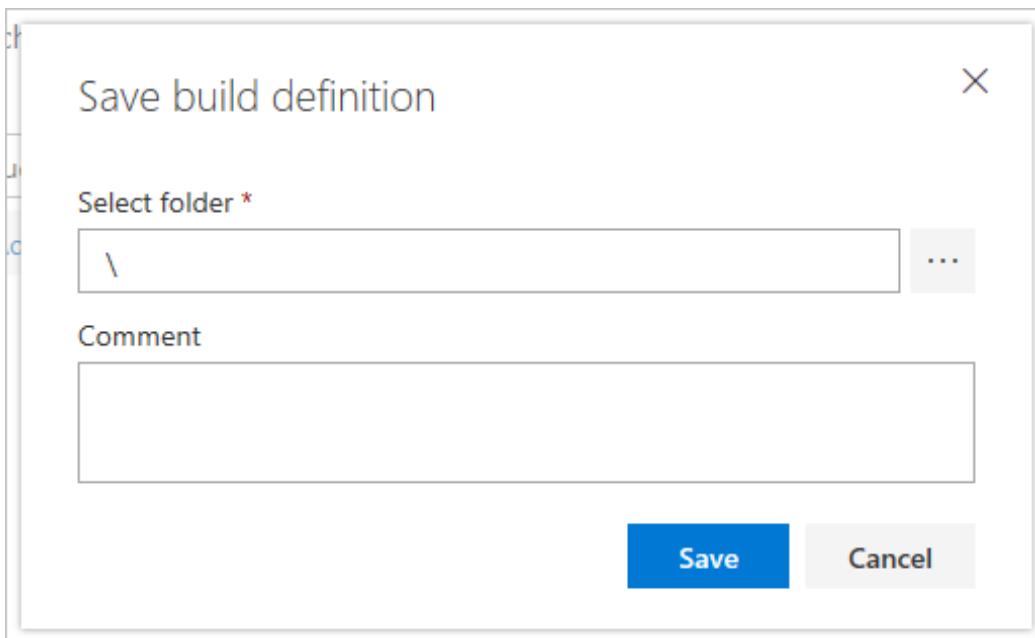
These settings cause a build to trigger when any change is pushed to the *master* branch of the GitHub repository. Continuous integration is tested in the [Commit changes to GitHub and automatically deploy to Azure](#) section.

39. Click the **Save & queue** button, and select the **Save** option:



Save button

40. The following modal dialog appears:

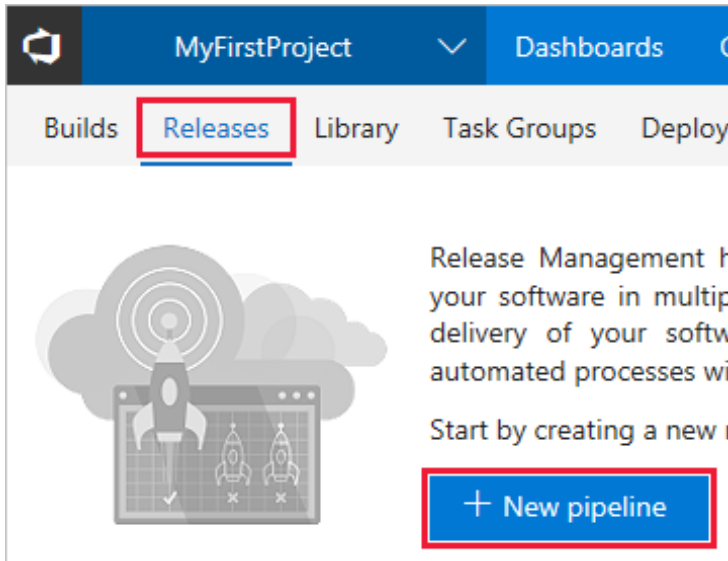


Save build definition - modal dialog

Use the default folder of \, and click the **Save** button.

Create the release pipeline

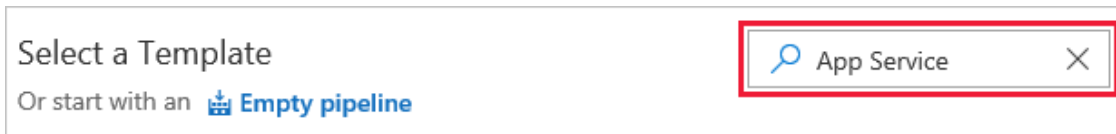
41. Click the **Releases** tab of your team project. Click the **New pipeline** button.



Releases tab - New definition button

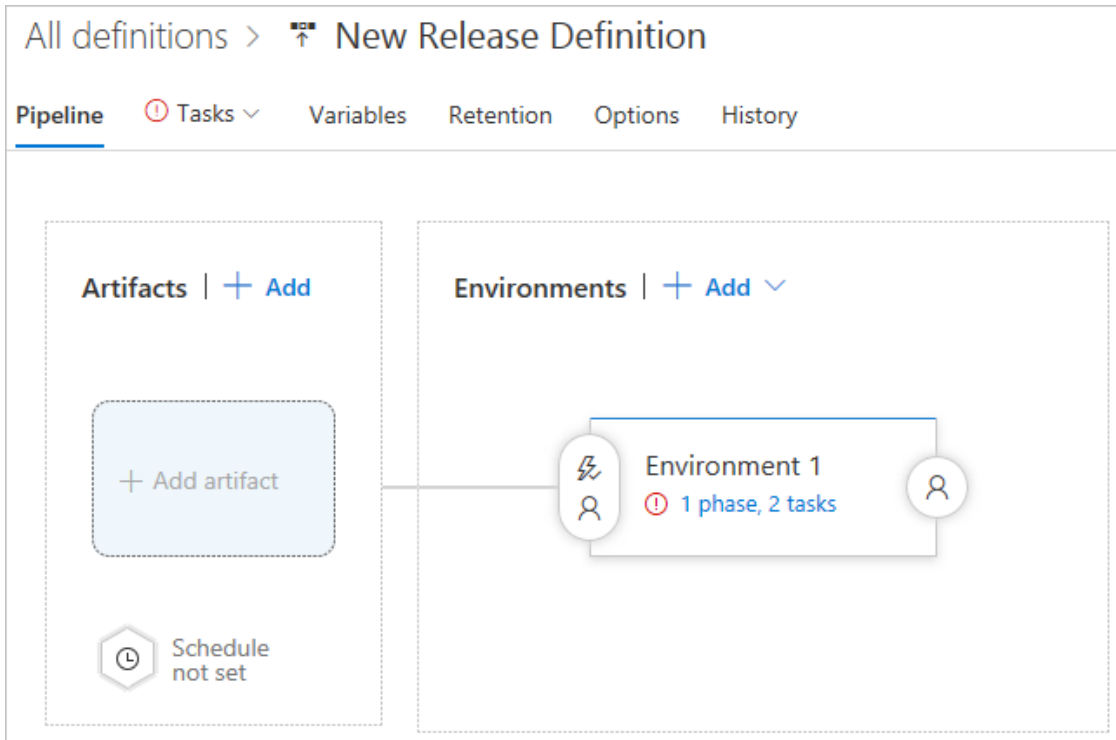
The template selection pane appears.

42. From the template selection page, enter *App Service* in the search box:



Release pipeline template search box

43. The template search results appear. Hover over the **Azure App Service Deployment with Slot** template, and click the **Apply** button. The **Pipeline** tab of the release pipeline appears.

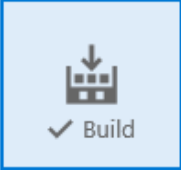


Release pipeline Pipeline tab


44. Click the **Add** button in the **Artifacts** box. The **Add artifact** panel appears:

Add artifact


Source type




✓ Build



Git



GitHub



Team Found...

[3 more artifact types](#) ▾

Project * ⓘ

MyFirstProject
▾

Source (Build definition) * ⓘ

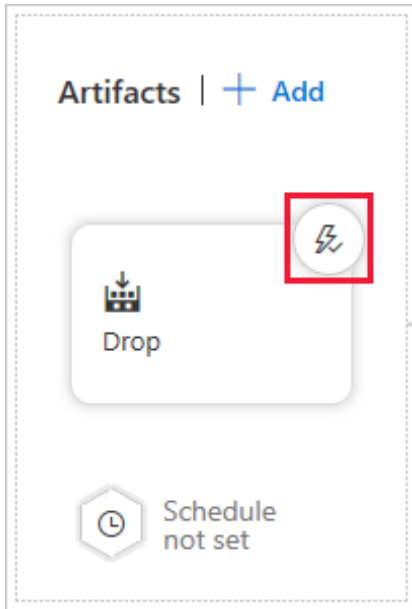
▾

ⓘ This setting is required.

Add

Release pipeline - Add artifact panel

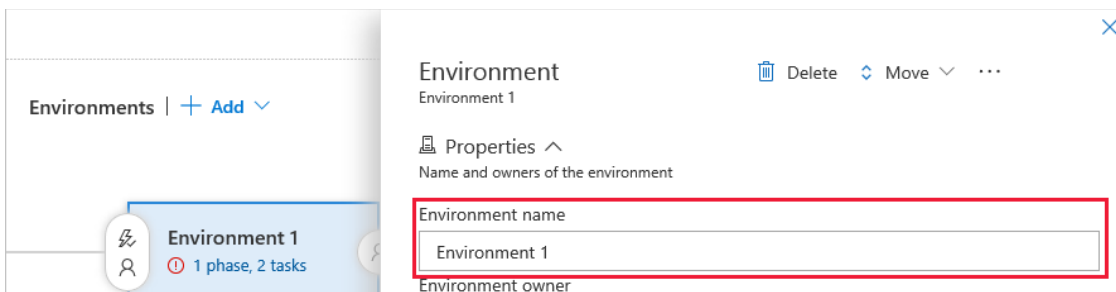
45. Select the **Build** tile from the **Source type** section. This type allows for the linking of the release pipeline to the build definition.
46. Select *MyFirstProject* from the **Project** drop-down.
47. Select the build definition name, *MyFirstProject-ASP.NET Core-CI*, from the **Source (Build definition)** drop-down.
48. Select *Latest* from the **Default version** drop-down. This option builds the artifacts produced by the latest run of the build definition.
49. Replace the text in the **Source alias** textbox with *Drop*.
50. Click the **Add** button. The **Artifacts** section updates to display the changes.
51. Click the lightning bolt icon to enable continuous deployments:



Release pipeline Artifacts - lightning bolt icon

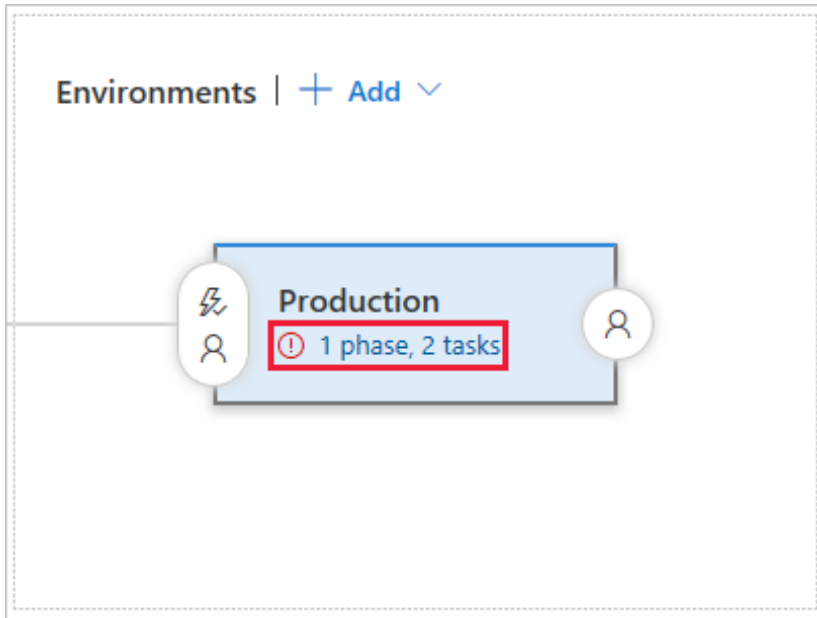
With this option enabled, a deployment occurs each time a new build is available.

52. A **Continuous deployment trigger** panel appears to the right. Click the toggle button to enable the feature. It isn't necessary to enable the **Pull request trigger**.
53. Click the **Add** drop-down in the **Build branch filters** section. Choose the **Build Definition's default branch** option. This filter causes the release to trigger only for a build from the GitHub repository's *master* branch.
54. Click the **Save** button. Click the **OK** button in the resulting **Save** modal dialog.
55. Click the **Environment 1** box. An **Environment** panel appears to the right. Change the *Environment 1* text in the **Environment name** textbox to *Production*.



Release pipeline - Environment name textbox

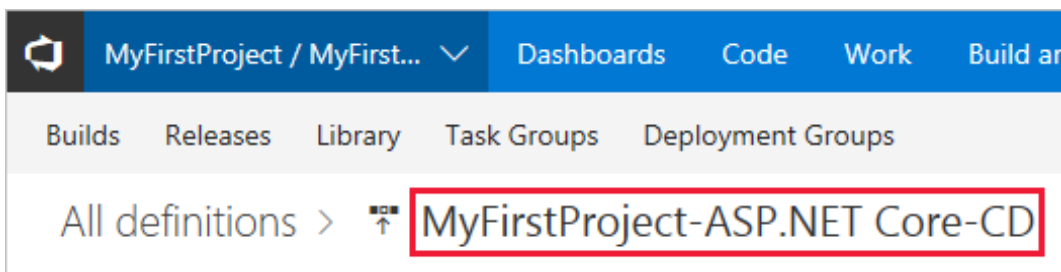
56. Click the **1 phase, 2 tasks** link in the **Production** box:



Release pipeline - Production environment link.png

The **Tasks** tab of the environment appears.

57. Click the **Deploy Azure App Service to Slot** task. Its settings appear in a panel to the right.
58. Select the Azure subscription associated with the App Service from the **Azure subscription** drop-down. Once selected, click the **Authorize** button.
59. Select *Web App* from the **App type** drop-down.
60. Select *mywebapp/* from the **App service name** drop-down.
61. Select *AzureTutorial* from the **Resource group** drop-down.
62. Select *staging* from the **Slot** drop-down.
63. Click the **Save** button.
64. Hover over the default release pipeline name. Click the pencil icon to edit it. Use *MyFirstProject-ASP.NET Core-CD* as the name.



Release pipeline name

65. Click the **Save** button.

Commit changes to GitHub and automatically deploy to Azure

66. Open *SimpleFeedReader.sln* in Visual Studio.

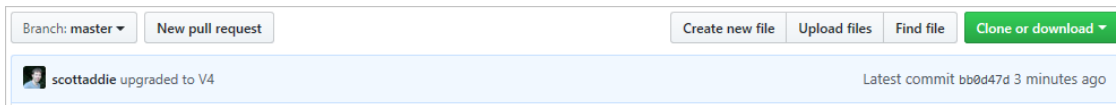
67. In Solution Explorer, open *Pages.cshtml*. Change `<h2>Simple Feed Reader - V3</h2>` to `<h2>Simple Feed Reader - V4</h2>`.
68. Press **Ctrl+Shift+B** to build the app.
69. Commit the file to the GitHub repository. Use either the **Changes** page in Visual Studio's *Team Explorer* tab, or execute the following using the local machine's command shell:

```
git commit -a -m "upgraded to V4"
```

70. Push the change in the *master* branch to the *origin* remote of your GitHub repository:

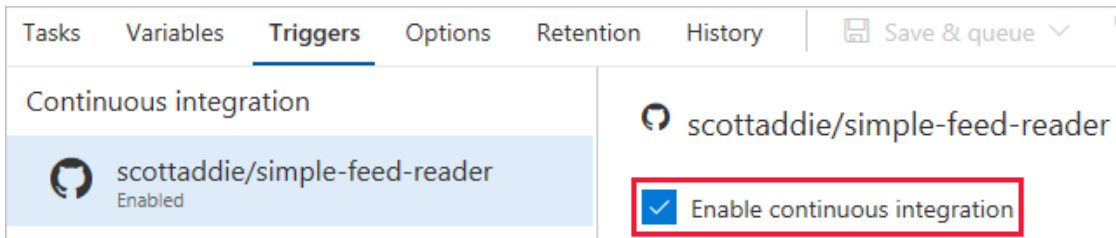
```
git push origin master
```

The commit appears in the GitHub repository's *master* branch:



GitHub commit in master branch

The build is triggered, since continuous integration is enabled in the build definition's **Triggers** tab:



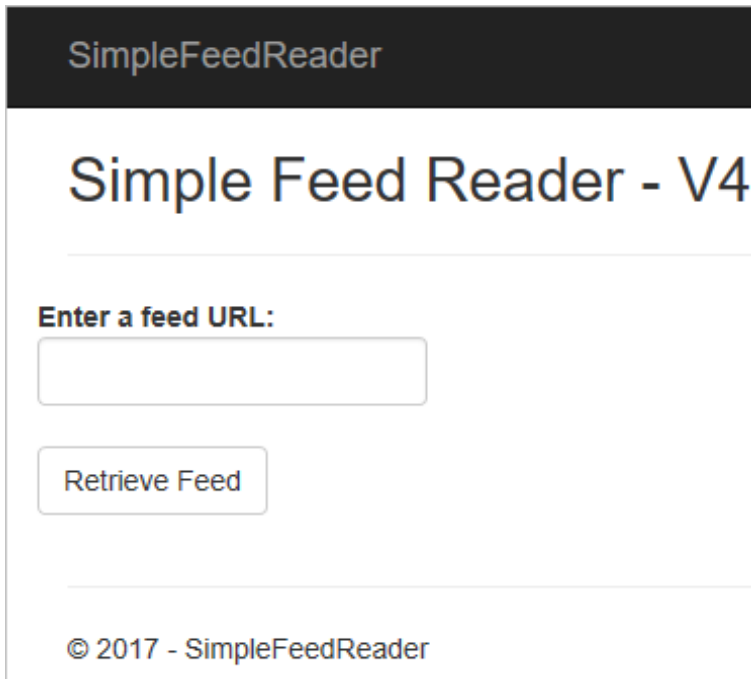
enable continuous integration

71. Navigate to the **Queued** tab of the **Build and Release > Builds** page in VSTS. The queued build shows the branch and commit that triggered the build:

Queued or running						
	Name	Definition name	Queue name	Source	Source version	Date queued
	20180614.1	MyFirstProject-ASP.NET Core-CI	Hosted VS2017	master	bb0d47d	a minute ago

queued build

72. Once the build succeeds, a deployment to Azure occurs. Navigate to the app in the browser. Notice that the "V4" text appears in the heading:



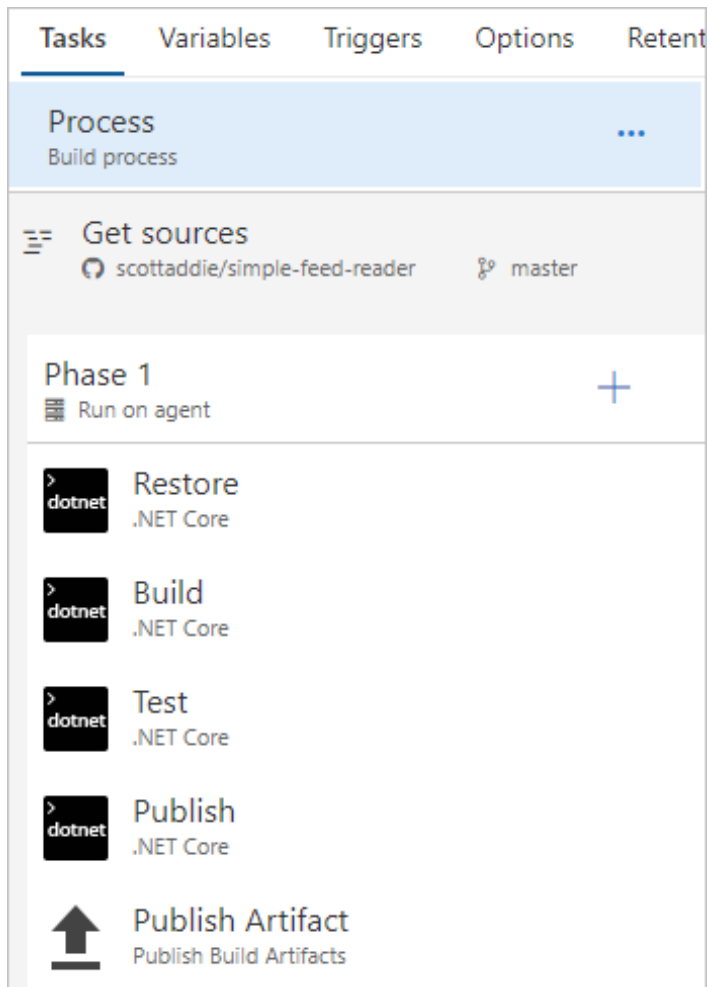
updated app

Examine the VSTS DevOps pipeline

Build definition

A build definition was created with the name *MyFirstProject-ASP.NET Core-CI*. Upon completion, the build produces a *.zip* file including the assets to be published. The release pipeline deploys those assets to Azure.

The build definition's **Tasks** tab lists the individual steps being used. There are five build tasks.



build definition tasks

73. **Restore** — Executes the `dotnet restore` command to restore the app's NuGet packages. The default package feed used is nuget.org.
74. **Build** — Executes the `dotnet build --configuration release` command to compile the app's code. This `--configuration` option is used to produce an optimized version of the code, which is suitable for deployment to a production environment. Modify the *BuildConfiguration* variable on the build definition's **Variables** tab if, for example, a debug configuration is needed.
75. **Test** — Executes the `dotnet test --configuration release --logger trx --results-directory <local_path_on_build_agent>` command to run the app's unit tests. Unit tests are executed within any C# project matching the `**/*Tests/*.csproj` glob pattern. Test results are saved in a `.trx` file at the location specified by the `--results-directory` option. If any tests fail, the build fails and isn't deployed.

Note - To verify the unit tests work, modify *SimpleFeedReader.Tests.cs* to purposefully break one of the tests. For example, change `Assert.True(result.Count > 0);` to `Assert.False(result.Count > 0);` in the *Returns_News_Stories_Given_Valid_Uri* method. Commit and push the change to GitHub. The build is triggered and fails. The build pipeline status changes to **failed**. Revert the change, commit, and push again. The build succeeds.

76. **Publish** — Executes the `dotnet publish --configuration release --output <local_path_on_build_agent>` command to produce a `.zip` file with the artifacts to be deployed. The `--output` option specifies the publish location of the `.zip` file. That location is specified by passing a [predefined variable](#) named `$(build.artifactstagingdirectory)`. That variable expands to a local path, such as `*c:_work\1`, on the build agent.
77. **Publish Artifact** — Publishes the `.zip` file produced by the **Publish** task. The task accepts the `.zip` file location as a parameter, which is the predefined variable `$(build.artifactstagingdirectory)`. The `.zip` file is published as a folder named `drop`.

Click the build definition's **Summary** link to view a history of builds with the definition:



build definition history

On the resulting page, click the link corresponding to the unique build number:

The image shows the 'Summary' page for a build definition. At the top, there are three tabs: 'Summary', 'History', and 'Deleted'. Below the tabs is a 'Details' section with the following information:

- Repository: [scottaddie/simple-feed-reader](#)
- Default queue: Hosted VS2017 | [Manage](#)
- Queue status: Enabled
- Last updated by: Scott Addie | Friday, May 25, 2018 3:23 PM

Below the details is a 'Queued & running' section with the text: 'No builds queued or running at the moment'.

Below that is a 'Recently completed' section with a table:

Build Number	Status	Branch	Author
#20180525.1	✓ succeeded	master	Scott Addie

Below the table is an 'Analytics' section with two metrics:

- Number of builds: 1
- Success rate: 100.00%

build definition summary page

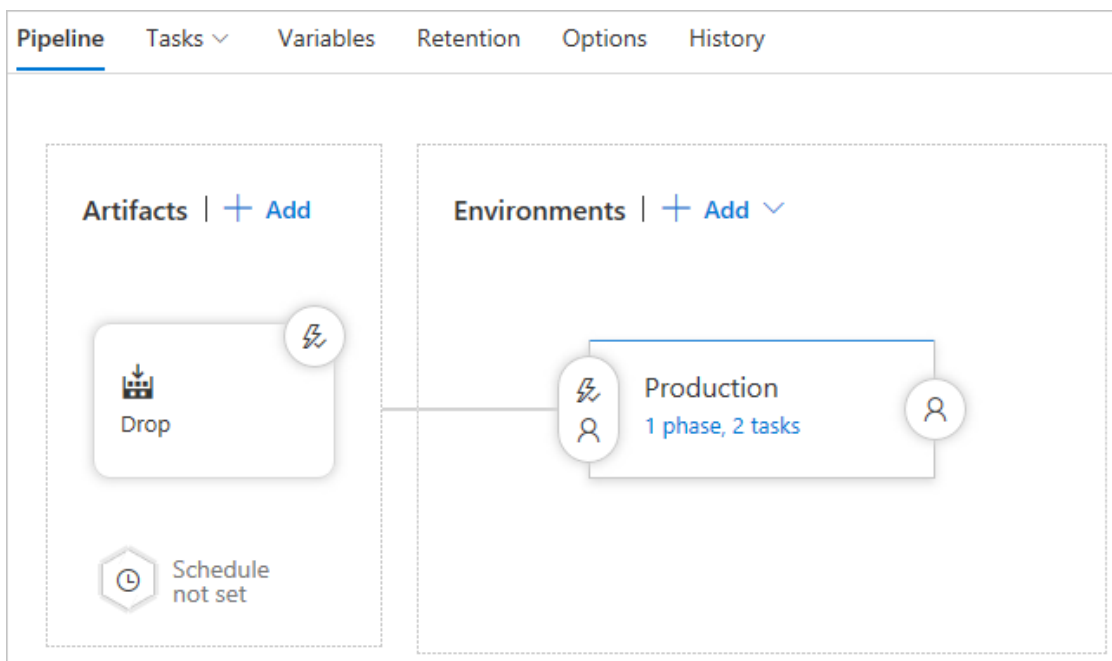
A summary of this specific build is displayed. Click the **Artifacts** tab, and notice the `drop` folder produced by the build is listed:

build definition artifacts - drop folder

Use the **Download** and **Explore** links to inspect the published artifacts.

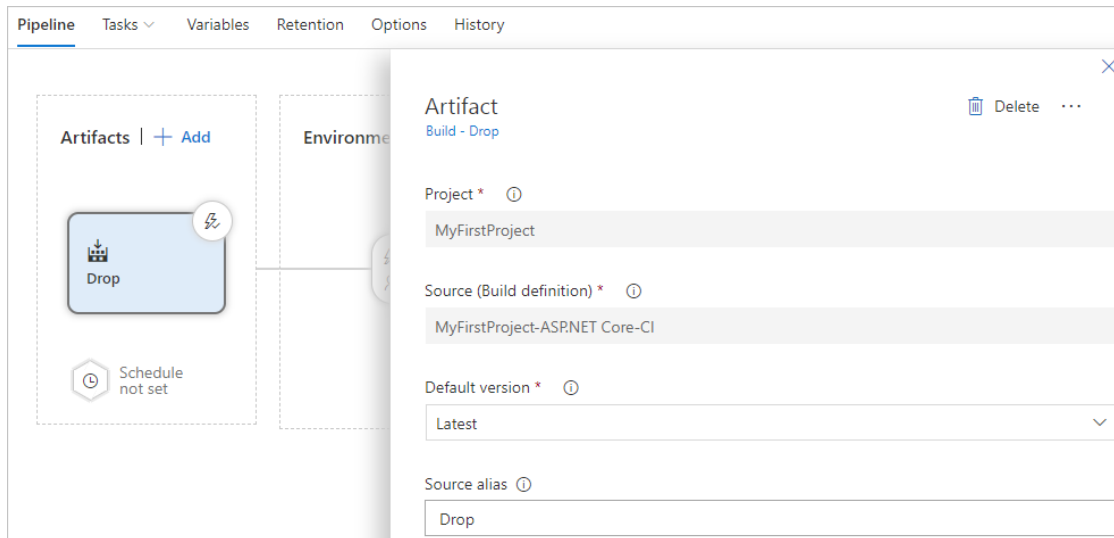
Release pipeline

A release pipeline was created with the name *MyFirstProject-ASP.NET Core-CD*:



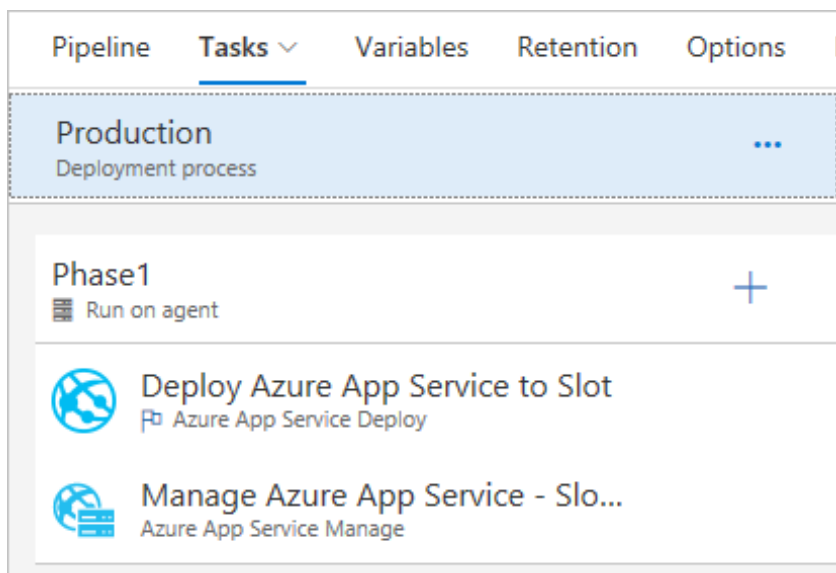
release pipeline overview

The two major components of the release pipeline are the **Artifacts** and the **Environments**. Clicking the box in the **Artifacts** section reveals the following panel:



release pipeline artifacts

The **Source (Build definition)** value represents the build definition to which this release pipeline is linked. The *.zip* file produced by a successful run of the build definition is provided to the *Production* environment for deployment to Azure. Click the *1 phase, 2 tasks* link in the *Production* environment box to view the release pipeline tasks:



release pipeline tasks

The release pipeline consists of two tasks: *Deploy Azure App Service to Slot* and *Manage Azure App Service - Slot Swap*. Clicking the first task reveals the following task configuration:

Azure App Service Deploy ⓘ

📁 Version

Display name *

Azure subscription * ⓘ | [Manage](#)

ⓘ

App type * ⓘ

App Service name * ⓘ ⓘ

Deploy to slot ⓘ

Resource group * ⓘ ⓘ

Slot * ⓘ ⓘ

release pipeline deploy task

The Azure subscription, service type, web app name, resource group, and deployment slot are defined in the deployment task. The **Package or folder** textbox holds the *.zip* file path to be extracted and deployed to the *staging* slot of the *mywebapp<unique_number>* web app.

Clicking the slot swap task reveals the following task configuration:

Azure App Service Manage ⓘ

Version 0.* ▾

Display name *
Manage Azure App Service - Slot Swap

Azure subscription * ⓘ | [Manage](#) ↗
Visual Studio Enterprise ▾ ↻

Action ⓘ
Swap Slots ▾

App Service name * ⓘ
mywebapp11857 ▾ ↻

Resource group * ⓘ
AzureTutorial ▾ ↻

Source Slot * ⓘ
staging ▾ ↻

Swap with Production ⓘ

Preserve Vnet ⓘ

release pipeline slot swap task

The subscription, resource group, service type, web app name, and deployment slot details are provided. The **Swap with Production** checkbox is checked. Consequently, the bits deployed to the *staging* slot are swapped into the production environment.

Additional reading

- [Build your ASP.NET Core app](#)
- [Build and deploy to an Azure Web App](#)
- [Define a CI build process for your GitHub repository](#)

Monitor and debug

Having deployed the app and built a DevOps pipeline, it's important to understand how to monitor and troubleshoot the app.

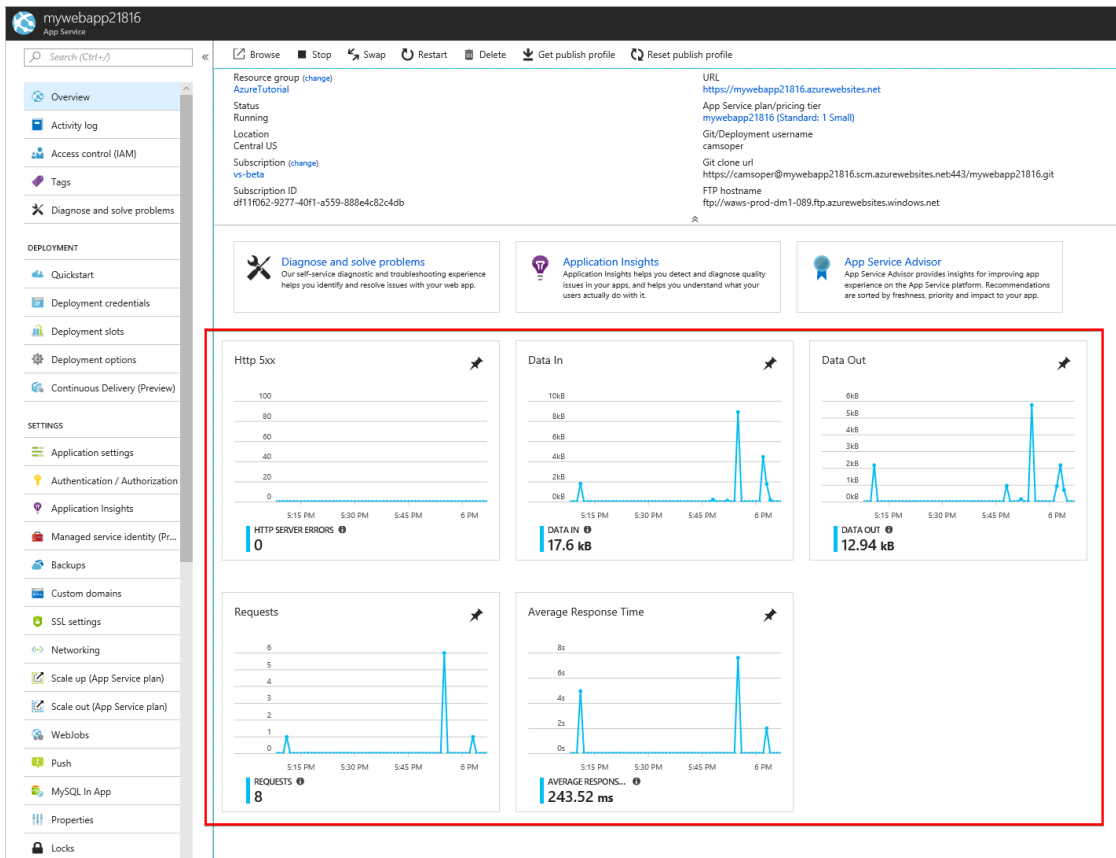
In this section, you'll complete the following tasks:

- Find basic monitoring and troubleshooting data in the Azure portal
- Learn how Azure Monitor provides a deeper look at metrics across all Azure services
- Connect the web app with Application Insights for app profiling
- Turn on logging and learn where to download logs
- Stream logs in real time
- Learn where to set up alerts
- Learn about remote debugging Azure App Service web apps.

Basic monitoring and troubleshooting

App Service web apps are easily monitored in real time. The Azure portal renders metrics in easy-to-understand charts and graphs.

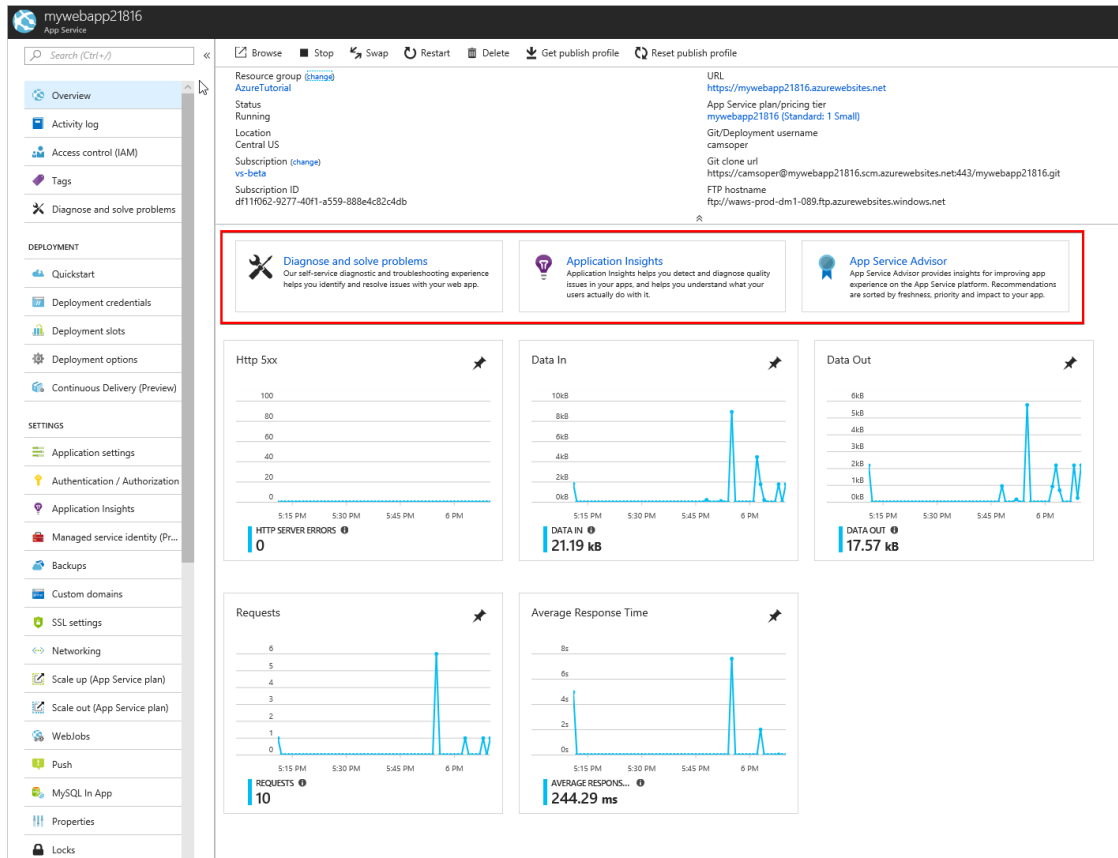
1. Open the [Azure portal](#), and then navigate to the *mywebapp<unique_number>* App Service.
2. The **Overview** tab displays useful “at-a-glance” information, including graphs displaying recent metrics.



Overview panel

- **Http 5xx:** Count of server-side errors, usually exceptions in ASP.NET Core code.
- **Data In:** Data ingress coming into your web app.
- **Data Out:** Data egress from your web app to clients.
- **Requests:** Count of HTTP requests.
- **Average Response Time:** Average time for the web app to respond to HTTP requests.

Several self-service tools for troubleshooting and optimization are also found on this page.



Self-service tools

- **Diagnose and solve problems** is a self-service troubleshooter.
- **Application Insights** is for profiling performance and app behavior, and is discussed later in this section.
- **App Service Advisor** makes recommendations to tune your app experience.

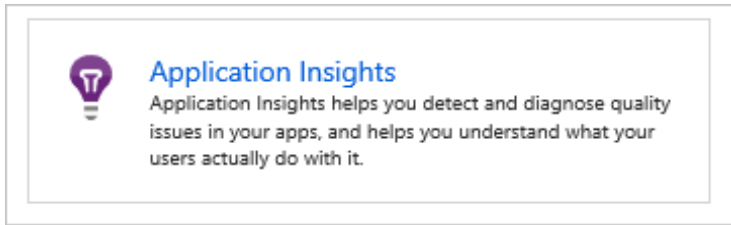
Advanced monitoring

Azure Monitor is the centralized service for monitoring all metrics and setting alerts across Azure services. Within Azure Monitor, administrators can granularly track performance and identify trends. Each Azure service offers its own **set of metrics** to Azure Monitor.

Profile with Application Insights


Application Insights is an Azure service for analyzing the performance and stability of web apps and how users use them. The data from Application Insights is broader and deeper than that of Azure Monitor. The data can provide developers and administrators with key information for improving apps. Application Insights can be added to an Azure App Service resource without code changes.

1. Open the **Azure portal**, and then navigate to the *mywebapp<unique_number>* App Service.
2. From the **Overview** tab, click the **Application Insights** tile.



Application Insights tile

3. Select the **Create new resource** radio button. Use the default resource name, and select the location for the Application Insights resource. The location doesn't need to match that of your web app.



Application Insights

Application Insights helps you detect and diagnose quality issues in your web apps and web services, and helps you actually do with it.
[Getting started with Application Insights monitoring](#)

Link your application to Application Insights

Select existing resource

my-blog	my-blog	South Central US
---------	---------	------------------

Create new resource

* New resource name ✓

* Location ▼

Instrument your application

* Runtime/Framework ▼

Code level diagnostics
Identify code that slowed down your web app and debug runtime exceptions with local variables

▼ Advanced Settings

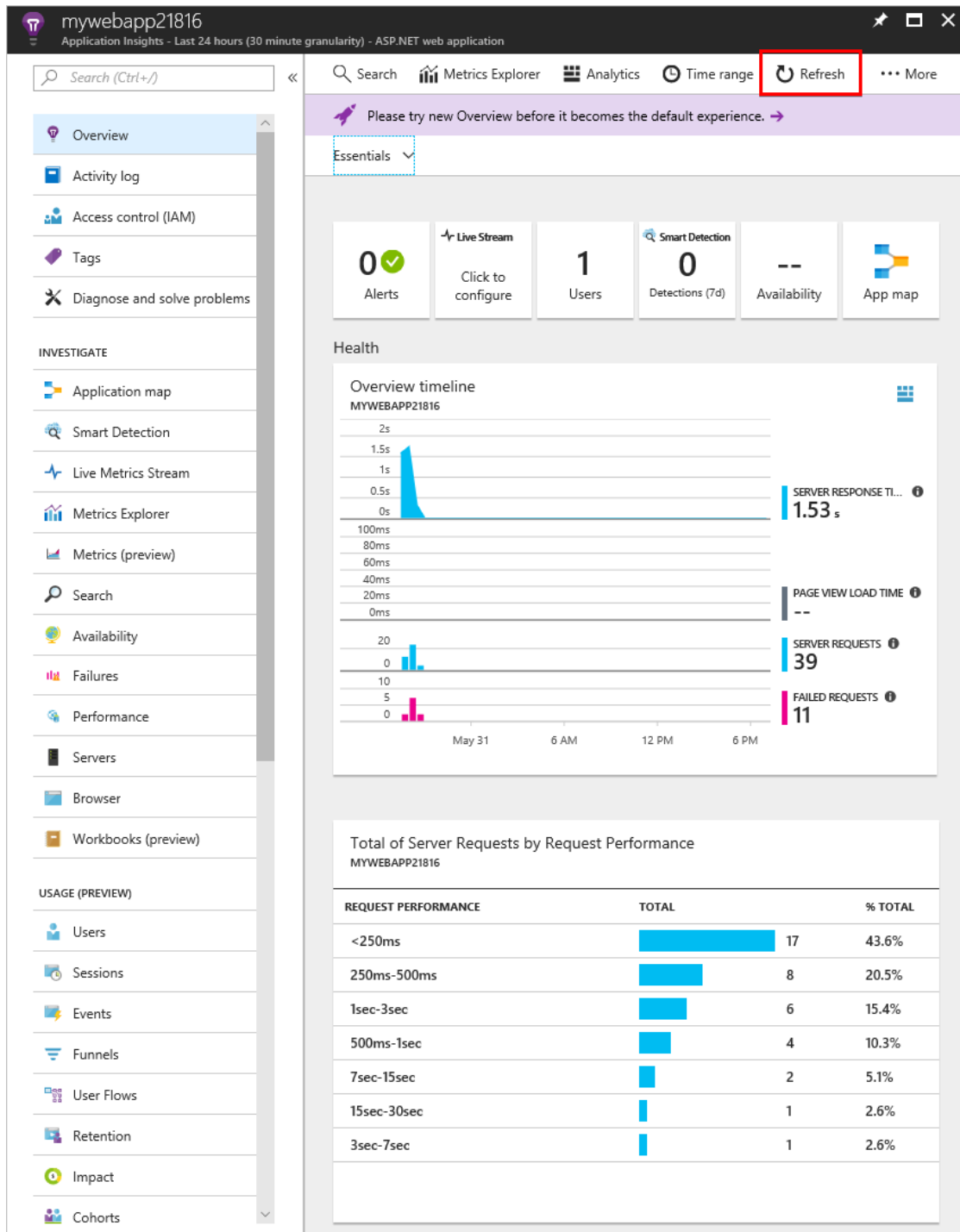
Application Insights setup

4. For **Runtime/Framework**, select **ASP.NET Core**. Accept the default settings.
5. Select **OK**. If prompted to confirm, select **Continue**.
6. After the resource has been created, click the name of Application Insights resource to navigate directly to the Application Insights page.

 **mywebapp21816** Application Insights resource is connected [Change](#)
[Troubleshooting information](#)

New Application Insights resource is ready

As the app is used, data accumulates. Select **Refresh** to reload the blade with new data.



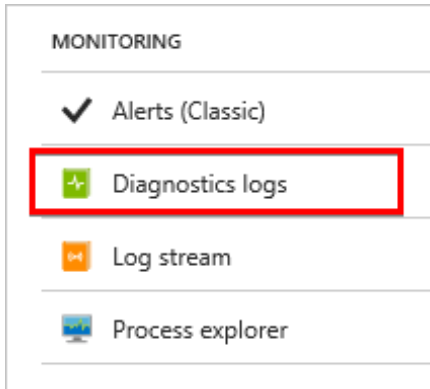
Application Insights overview tab

Application Insights provides useful server-side information with no additional configuration. To get the most value from Application Insights, [instrument your app with the Application Insights SDK](#). When properly configured, the service provides end-to-end monitoring across the web server and browser, including client-side performance. For more information, see the [Application Insights documentation](#).

Logging

Web server and app logs are disabled by default in Azure App Service. Enable the logs with the following steps:

1. Open the [Azure portal](#), and navigate to the *mywebapp<unique_number>* App Service.
2. In the menu to the left, scroll down to the **Monitoring** section. Select **Diagnostics logs**.



Diagnostic logs link

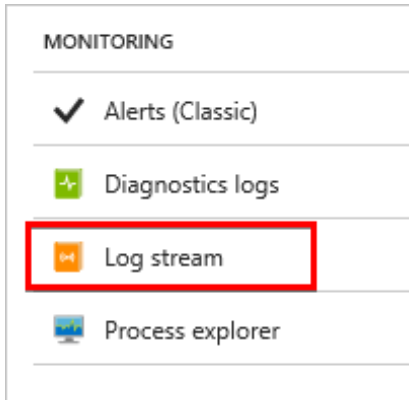
3. Turn on **Application Logging (Filesystem)**. If prompted, click the box to install the extensions to enable app logging in the web app.
4. Set **Web server logging** to **File System**.
5. Enter the **Retention Period** in days. For example, 30.
6. Click **Save**.

ASP.NET Core and web server (App Service) logs are generated for the web app. They can be downloaded using the FTP/FTPS information displayed. The password is the same as the deployment credentials created earlier in this guide. The logs can be [streamed directly to your local machine with PowerShell or Azure CLI](#). Logs can also be [viewed in Application Insights](#).

Log streaming

App and web server logs can be streamed in real time through the portal.

1. Open the [Azure portal](#), and navigate to the *mywebapp<unique_number>* App Service.
2. In the menu to the left, scroll down to the **Monitoring** section and select **Log stream**.



Log stream link

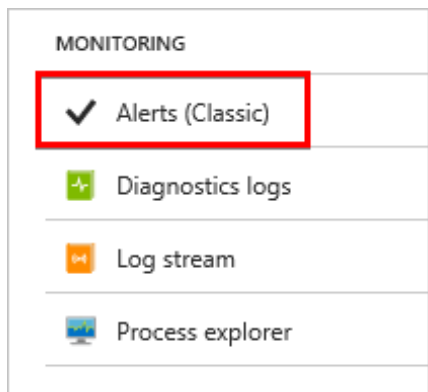
Logs can also be [streamed via Azure CLI or Azure PowerShell](#), including through the Cloud Shell.

Alerts

Azure Monitor also provides [real time alerts](#) based on metrics, administrative events, and other criteria.

Note: Currently alerting on web app metrics is only available in the Alerts (classic) service.

The [Alerts \(classic\) service](#) can be found in Azure Monitor or under the **Monitoring** section of the App Service settings.



Alerts (classic) link

Live debugging

Azure App Service can be [debugged remotely with Visual Studio](#) when logs don't provide enough information. However, remote debugging requires the app to be compiled with debug symbols. Debugging shouldn't be done in production, except as a last resort.

Conclusion

In this section, you completed the following tasks:

- Find basic monitoring and troubleshooting data in the Azure portal
- Learn how Azure Monitor provides a deeper look at metrics across all Azure services
- Connect the web app with Application Insights for app profiling
- Turn on logging and learn where to download logs
- Stream logs in real time
- Learn where to set up alerts
- Learn about remote debugging Azure App Service web apps.

Additional reading

- [Troubleshoot ASP.NET Core on Azure App Service](#)
- [Common errors reference for Azure App Service and IIS with ASP.NET Core](#)
- [Monitor Azure web app performance with Application Insights](#)
- [Enable diagnostics logging for web apps in Azure App Service](#)
- [Troubleshoot a web app in Azure App Service using Visual Studio](#)
- [Create classic metric alerts in Azure Monitor for Azure services - Azure portal](#)

Next steps

In this guide, you created a DevOps pipeline for an ASP.NET Core sample app. Congratulations! We hope you enjoyed learning to publish ASP.NET Core web apps to Azure App Service and automate the continuous integration of changes.

Beyond web hosting and DevOps, Azure has a wide array of Platform-as-a-Service (PaaS) services useful to ASP.NET Core developers. This section gives a brief overview of some of the most commonly used services.

Storage and databases

[Redis Cache](#) is high-throughput, low-latency data caching available as a service. It can be used for caching page output, reducing database requests, and providing ASP.NET session state across multiple instances of an app.

[Azure Storage](#) is Azure's massively scalable cloud storage. Developers can take advantage of [Queue Storage](#) for reliable message queuing, and [Table Storage](#) is a NoSQL key-value store designed for rapid development using massive, semi-structured data sets.

[Azure SQL Database](#) provides familiar relational database functionality as a service using the Microsoft SQL Server Engine.

[Cosmos DB](#) globally distributed, multi-model NoSQL database service. Multiple APIs are available, including SQL API (formerly called DocumentDB), Cassandra, and MongoDB.

Identity

[Azure Active Directory](#) and [Azure Active Directory B2C](#) are both identity services. Azure Active Directory is designed for enterprise scenarios and enables Azure AD B2B (business-to-business) collaboration, while Azure Active Directory B2C is intended business-to-customer scenarios, including social network sign-in.

Mobile

[Notification Hubs](#) is a multi-platform, scalable push-notification engine to quickly send millions of messages to apps running on various types of devices.

Web infrastructure

[Azure Container Service](#) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized apps without container orchestration expertise.

[Azure Search](#) is used to create an enterprise search solution over private, heterogenous content.

[Service Fabric](#) is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers.