4
Continuous Integration and Build Automation

As a developer compiling code and running unit tests gives you assurance that your code changes haven't had an impact on the existing codebase. Integrating your code changes into the source control repository enables other to validate their changes with yours. As a best practice teams integrate into the shared repository several times a day to reduce the risk of introducing breaking changes or worst overwriting each other's.

Continuous Integration (**CI**) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is verified by an automated build, allowing teams to detect problems early.

The Automated build running as part of the CI process is often referred to as the CI build. While there isn't a clear definition of what the CI build should do, at the very minimum it is expected to compile code and run unit tests. Running the CI build on a non-developer isolated workspace helps identify dependencies that may otherwise go unnoticed late into the release process. We can talk endlessly about the benefits of CI; the key here is that it enables you to have potentially deployable software at all times.

Deployable software is the most tangible asset to customers.

Moving from concept to application, in this chapter we'll learn how to leverage the build tooling in Azure DevOps Server to set up a quality focused Continuous Integration process.

In this chapter, we will cover the following recipes:

- Configuring one build definition for all branches of a Git repository
- Reflecting the branch quality in the build name
- Using web deploy to create a package in an ASP.NET build pipeline
- Organizing build output into separate folders
- Configuring assembly version info in build pipelines
- Setting up a build pipeline for a .NET core application
- Setting up a build pipeline for Node.js application
- Setting up a build pipeline for your database projects
- Integrating SonarQube in build pipelines to manage technical debt

Configuring one build definition for all branches of a Git repository

The Git branching model and pull request workflow makes it so easy to manage the flow of code that you will get accustomed to creating a topic branch for each new item of work. **Continuous Integration** is table stakes for any organization looking to move into a DevOps way of working. Associating a Continuous Integration flow with every new Git topic branch you create can be cumbersome, as you'll need to create a new build definition for each Git branch.

This becomes an operational nightmare if the topic branches are short-lived. In this recipe, we'll learn how to use one build definition to build all your Git branches in a team project.

Getting ready

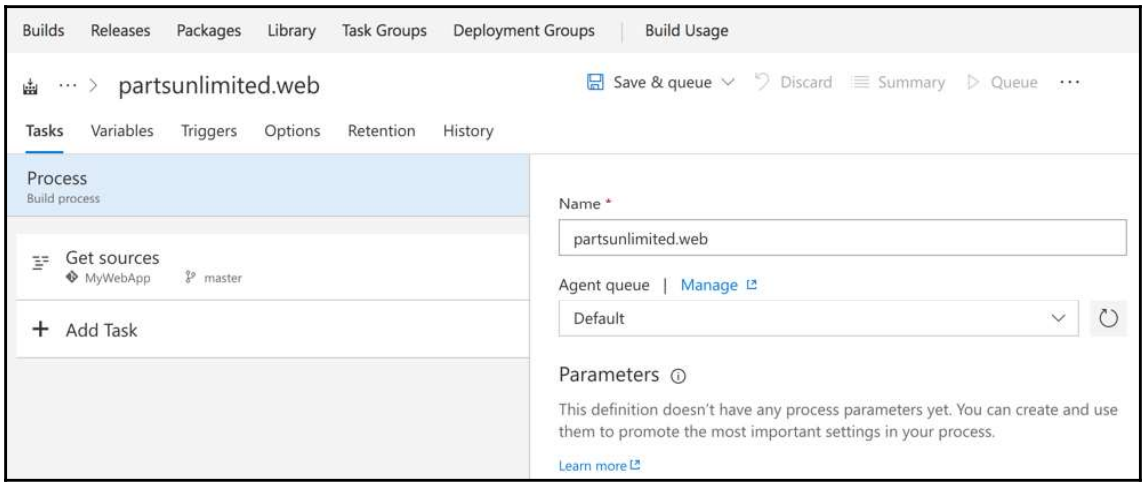
You need to be a member of the build administrator group in the team project. The build administrator group gives you permissions to administer build resources. Members can manage test environments, create test runs, and manage builds.

Create the following Git branches in your team project:

- **master**: Mainline for production
- **develop**: Integration for all features
- **feature/myFeature-1**: Feature development branch

How to do it...

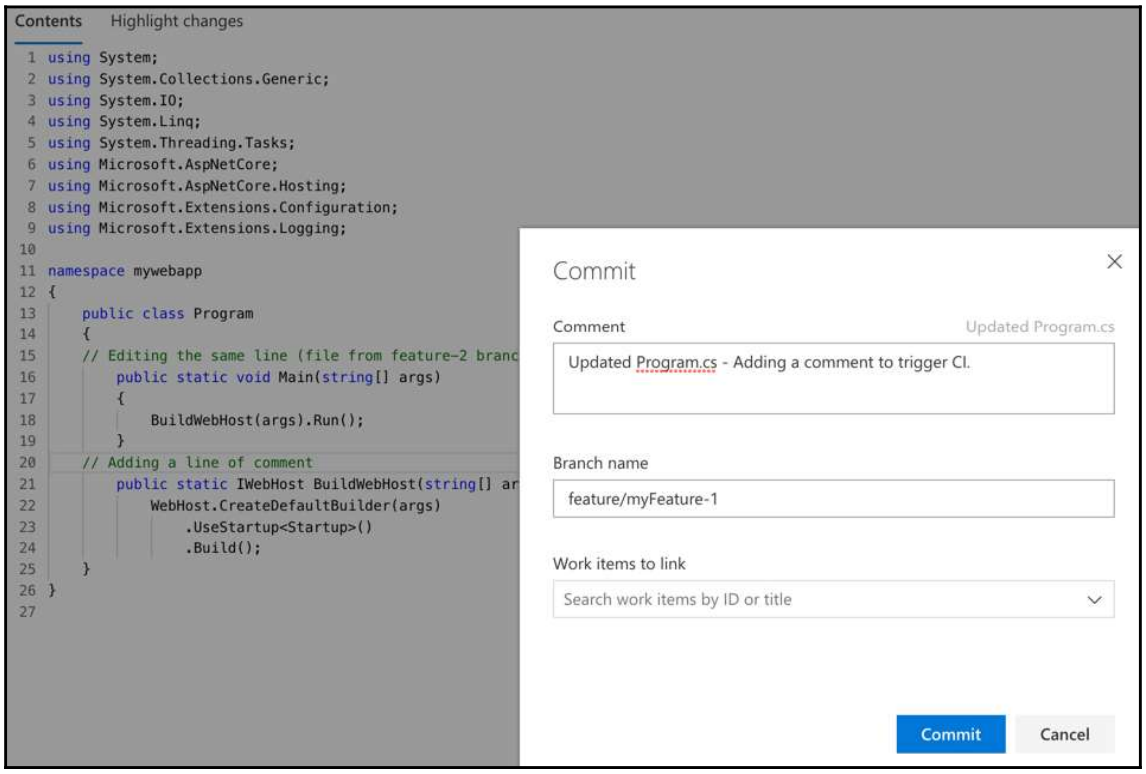
1. Navigate to the **Builds** and **Releases** hub in the parts unlimited team portal. From the **Builds** page, click **New** to create a new build definition.
2. Select the empty process to start with an empty build definition.
3. Name the build definition `partsunlimited.web`, choose the **Default** agent queue, and save the build definition, as shown in the following screenshot:



You can optionally add multiple branches using the **+ Add** link if you only intend to configure a build definition for multiple Git branches and not all of them.

How it works...

To better understand how this works, in this section we'll test the configuration and go through its implementation. Let's start by testing that the build definition is correctly configured to work against all branches in the `MyWebApp` Git repository. Navigate to the code hub, open the `MyWebApp` Git repository, and select the `feature/myFeature-1` branch. Edit the `Program.cs` file by adding a comment. Commit the changes to trigger the CI build for this branch:



See also

This approach can be extended to link one build definition to a release definition to release multiple branches into an environment. By using release artifact filters, you can lock down the topic branches to dev test environments only.

Reflecting the branch quality in the build name

Most software changes evolve from an alpha release quality to a beta release quality before they are ready to be shipped. This is often reflected in how the code moves between Git branches. Builds coming out of a topic branch where the change is still being matured are mostly alpha quality, while a first cut of the develop branch (as the changes are being integrated) where you are still soliciting feedback may be classed as beta quality before it's moved up to master, from where you tend to do production quality releases. In this recipe, we'll learn how to use the name of the branch to flag the quality of the build by appending it to the build name.

Getting ready

This is an extension to the *Configuring one build definition for all branches of a git repository* recipe. If you haven't already, configure a build definition to trigger all branches for the `MyWebApp` Git repository.

How to do it...

1. Navigate to the build view in the parts unlimited team portal and edit the `partsunlimited.web` build definition.In the **Options** tab, change the **Build number format** to `0.1.$(DayOfYear)$(Rev:.r)`, as shown in the following screenshot:

How it works...

Let's look at what the script is doing. The build exposes a number of predefined variables. This includes a combination of system variables used internally by the build system itself and helper variables that can be used to manage the build workflow. A list of all the pre-defined variables can be found here: `https://docs.microsoft.com/en-us/vsts/build-release/concepts/definitions/build/variables?view=vststabs=batch`.

The value of the `BUILD_SOURCEBRANCHNAME` build variable is read to identify if the source branch that triggered the build definition is `develop`. If so, the `BUILD_BUILDNUMBER` build variable is postfixed with `-beta`. For all other source branches (except master), the build number is updated as `-alpha`. However, in order to update the build variable, the build system uses the following format:

```
Write-Output "##vso[task.setvariable
variable=build.variablename;issecret=bool]new value"
```

The statement is a simplified implementation of this format to update the build variable that's used for the build number:

```
Write-Output ("##vso[build.updatebuildnumber]" + $env:BUILD_BUILDNUMBER+"-
alpha")
```

Using web deploy to create a package in an ASP.NET build pipeline

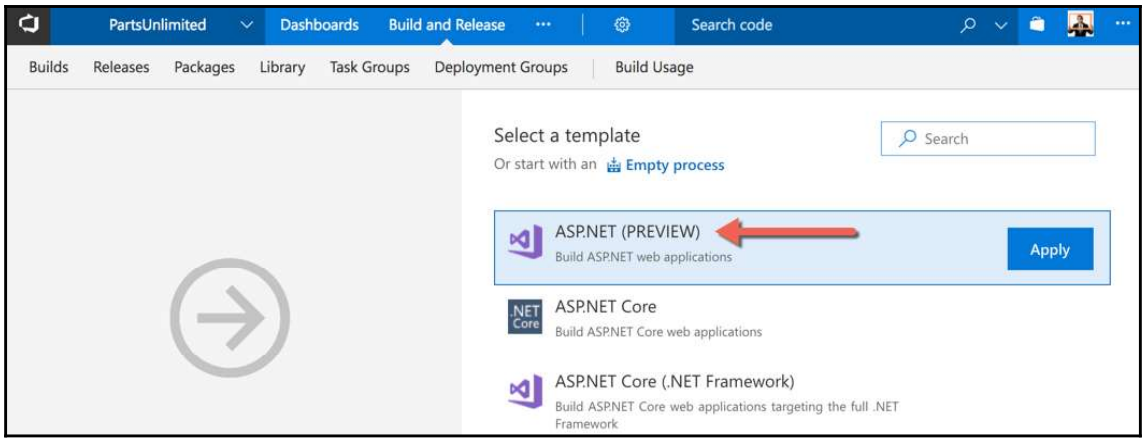
The build system in Azure DevOps Server ships a set of pre-canned build templates with all the necessary build tasks and configuration to help you get off the ground without having to learn how the build system works. If you are creating an application that uses the web project type in Visual Studio, then you'll be delighted to know that there is a build template you can apply to set up a build pipeline for your web application. In this recipe, we'll use the ASP.NET build template to create a build pipeline. In addition to building and testing the web app, this pipeline also creates a web deploy package that can be used to deploy to any web server, including Azure hosted app services.

Getting ready

Create a new Git repository—`MyModernWebApp`—in the parts unlimited team project. Create a new ASP.NET MVC project using Visual Studio and commit the changes to the master branch in the newly created Git repository.

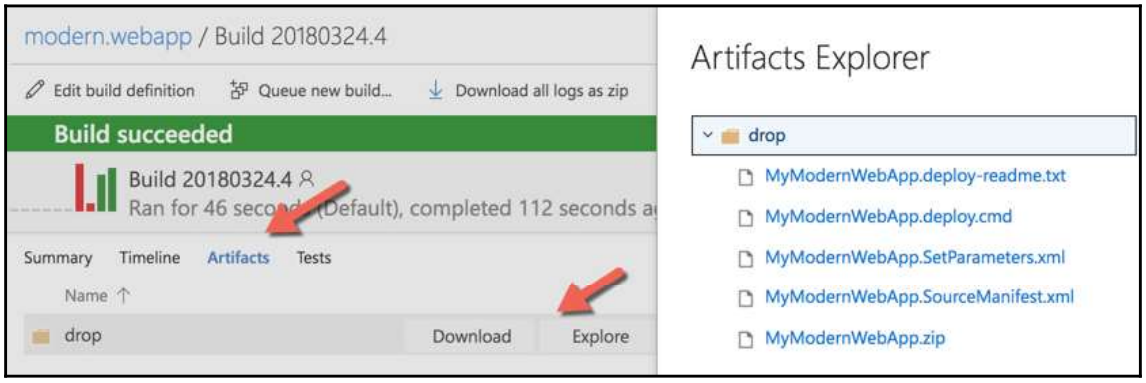
How to do it...

1. Navigate to the build view in the **PartsUnlimited** team portal. Create a new build definition by clicking the **+ New** button. This will show you a list of all the pre-canned build templates.
2. From the featured section, apply the ASP.NET build template, as shown in the following screenshot:



3. Name the build `modern.webapp`, select the default agent queue, and save the build definition. As you look around in the build definition, you'll see that the pre-canned pipeline has all the necessary tasks to restore the NuGet packages, as well as build, test, and publish a build artifact. The NuGet restore build task is configured to execute all `.sln` files from the source. The test step, on the other hand, is configured to operate on all DLL files that can be found in the `bin` folder using the `*test*.dll` convention and the publish artifact step publishes the `bin` folder of all projects associated to the solution. The build solution step also has a set of MSBuild arguments. It is these MSBuild arguments that trigger the generation of the Web Deploy package.

6. Click on the **Artifacts** section to see the web deploy package generated by this build execution:



How it works...

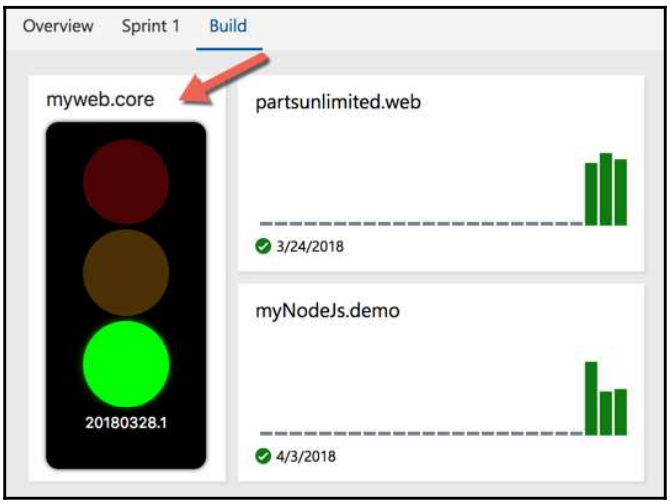
MSBuild ships with a set of command-line switches: `/p:` simply means property of MSBuild. The following MSBuild properties are used in the build step of the pipeline to create the web deploy package, these are explained in the section below:

```
/p:DeployOnBuild=true /p:WebPublishMethod=Package
/p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true
/p:PackageLocation="$(build.artifactstagingdirectory)\\"
```

- `DeployOnBuild`: This property is used to signal that the web project needs to be packaged in this build.
- `WebPublishMethod`: This property ensures that the output of the publish method is a package. This property supports other publish methods such as publishing to the filesystem or elsewhere using MSDeploy.
- `PackageAsSingleFile`: This property is used to signal that the package be zipped up into a single output file.
- `SkipInvalidConfigurations`: This tells the build engine to generate one or more warning if the build encounters an invalid configuration.
- `PackageLocation`: This takes the path where the package needs to be generated. We're using the default build variable to signal that the package should be copied into the build artifact staging directory, so that the next step in the build pipeline can pick the package from this location and publish the build artifact and attach it to the build.

There's more...

The Azure DevOps Server marketplace features the Build Traffic Lights extension: `https://marketplace.visualstudio.com/items?itemName=4tecture.BuildTrafficLights`. This free extension, developed by **4tecture**, allows you to add build traffic lights to your dashboard to visualize the state of a specific build definition and its builds. Continuous Integration is the foundation to Continuous Delivery, and showing the visibility of the CI pipeline on the team dashboard is a good way to encourage people to maintain a healthy CI pipeline:



Organizing build output into separate folders

In the DevOps way of working, teams are encouraged to adopt the right tools and practices earlier in the development lifecycle to minimize waste later. While the pre-canned build templates make it really easy to get started with build pipelines, the generic configuration bloats the build artifact and adds folders that you don't necessarily care about. The ones that you do care about are folded into multiple hierarchies. While it isn't necessarily a problem immediately, when you start to consume the build output in release pipelines, much of the release pipeline effort is spent in organizing the build output correctly. In the spirit of pushing more software development activities left into the lifecycle and minimizing waste, let's see how easy it is to organize the build output into relevant folders from the outset.

Getting ready

In this section we'll go through the pre-requisites for this recipe:

1. Extend the `MyModernWebApp` solution by adding two new projects of type (.NET Standard) console application and calling the first one `ExecutionEngine.Service` and the other one `MessagingEngine.Service`.
2. Commit the changes and sync them up to the origin/master in the `MyModernWebApp` Git repository:

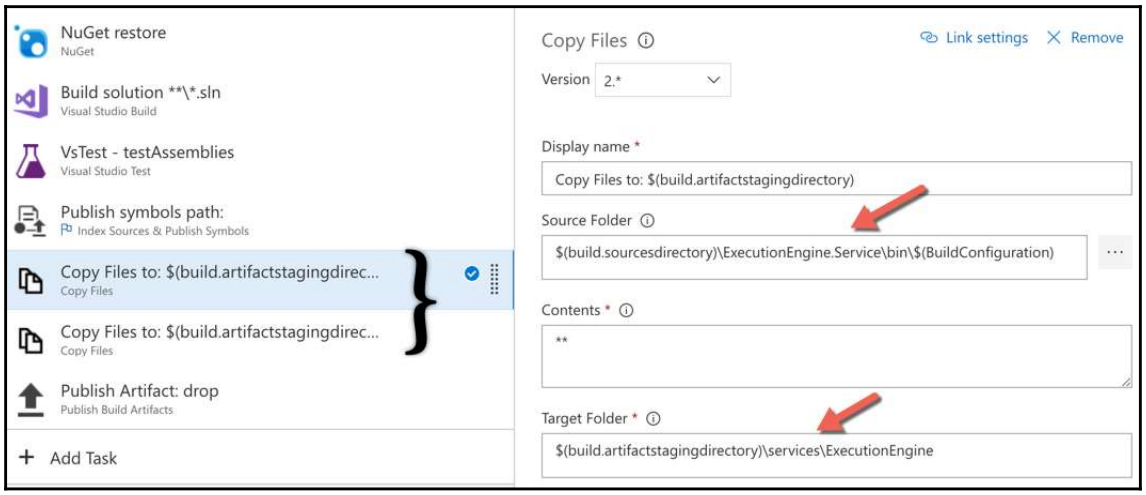
3. In the build view, click the **+ New** button to create a new build pipeline using the .NET Desktop pre-canned template.
4. Choose the default agent queue, name the build definition `modern.app.framework`, and queue a new build using this definition.

When the build completes, look at the build artifact: you'll see that the build output has uploaded the test project as an artifact. The build output of `ExecutionEngine.Service` and `MessagingEngine.Service` is tucked into `bin/release` folders.

Now that you have a sense of the problem, let's see how easy it is to take back control of the build output.

How to do it...

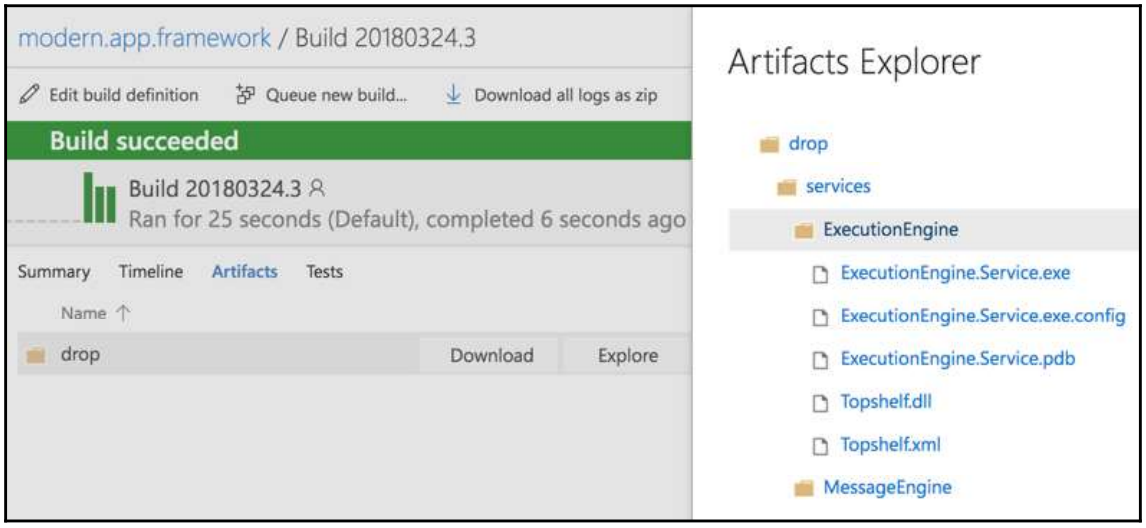
1. Navigate to and edit the `modern.app.framework` build definition.
2. Instead of overloading just one copy step to copy everything, we'll use multiple copy steps. The source folder location needs to be fully qualified to the exact location path from where the binaries need to be copied:



How it works...

Queue a build for the `modern.app.framework` definition; you'll now notice that the build output is a lot more organized. This has been done by removing the generic copy step and replacing it with two purposeful copy steps that fully qualify the source folder location and the target folder location. As a result, the test project DLL files haven't been uploaded as a build artifact.

The two service projects are nicely organized under the services project without being cryptically folded under the `bin/release` configuration:



Configuring assembly version info in build pipelines

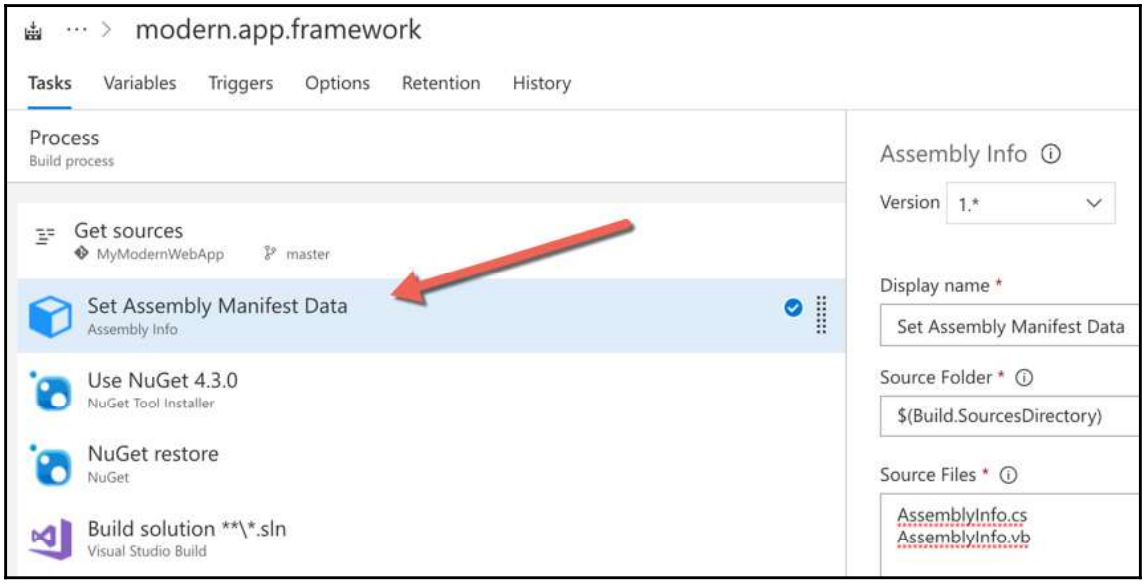
Azure DevOps Server provides a high level of traceability that makes it really easy to track builds generated from a build definition through to pull request, to code changes, and finally, back to work items. This traceability is, however, lost at the point when the binaries are generated through the build. Wouldn't it be great if you could look at the binaries deployed in an environment and identify the build they originated from? This could prove to be really useful when testing for regression issues. You can also take it a step further and display the binary version in the application, so when users log issues against your application they can also report the version of the application they are seeing the issues in. In this recipe, we'll learn how to configure the build number in the assemblies generated through a build pipeline.

Getting ready

The marketplace features the **Assembly Info** extension. This open source task, created by Bleddyn Richards, allows you to set assembly information such as version, copyright, trademark, and so on, right from within the build pipeline. Install the assembly info extension in your team project collection: `https://marketplace.visualstudio.com/items?itemName=bleddynrichards.Assembly-Info-Task`.

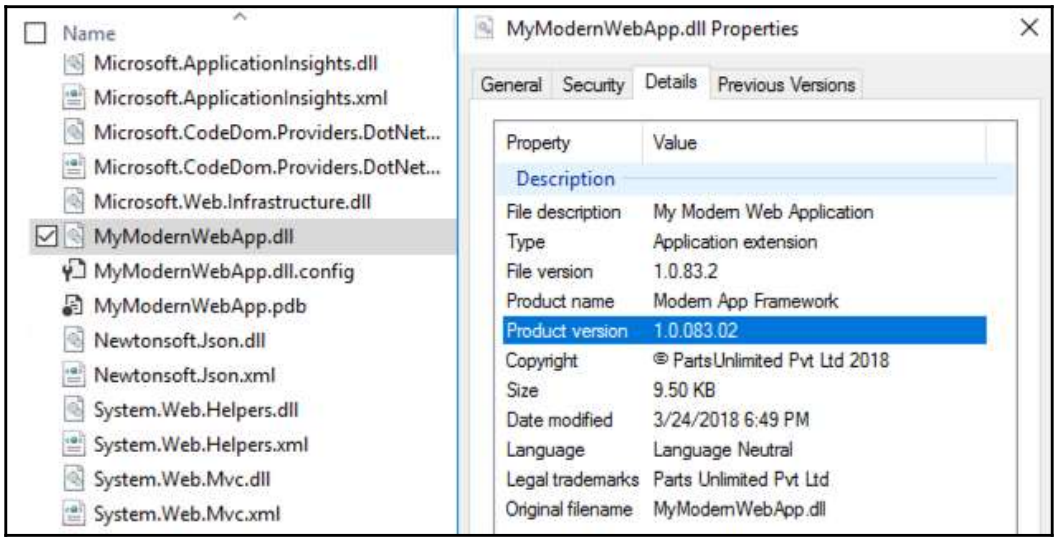
How to do it...

1. Navigate to the build view in the parts unlimited team portal and edit the previously created `modern.webapp` build definition.
2. Click **+ Add** to add the newly installed Assembly Info task into the `modern.webapp` build pipeline:



How it works...

1. Trigger a new build and wait for it to complete execution.
2. Download the build artifact and see the assembly property. This should correctly reflect the configuration specified by you in the Assembly Info task:



3. The Assembly Info task exposes the following fields via the build task. The following table shows you how these map back to the attributes in the `AssemblyInfo` file:

Field	Attribute	Function
Title	AssemblyTitle	Provides a friendly name for the assembly
Product	AssemblyProduct	Provides the product information for the assembly
Description	AssemblyDescription	Provides a short description that summarizes the nature and purpose of the assembly
Company	AssemblyCompany	Provides the company name for the assembly
Copyright	AssemblyCopyright	Provides the assembly or product copyright information
Trademark	AssemblyTrademark	Provides the assembly or product trademark information
Culture	AssemblyCulture	Provides information on what culture the assembly supports

Field	Attribute	Function
Configuration	AssemblyConfiguration	Provides the build configuration for the assembly, such as debug or release
Version number	AssemblyVersion	Provides an assembly version for the application
File version number	AssemblyFileVersion	Provides a file version for the application
Informational version	AssemblyInformationalVersion	Provides a text version for the application

Setting up a build pipeline for a .NET core application

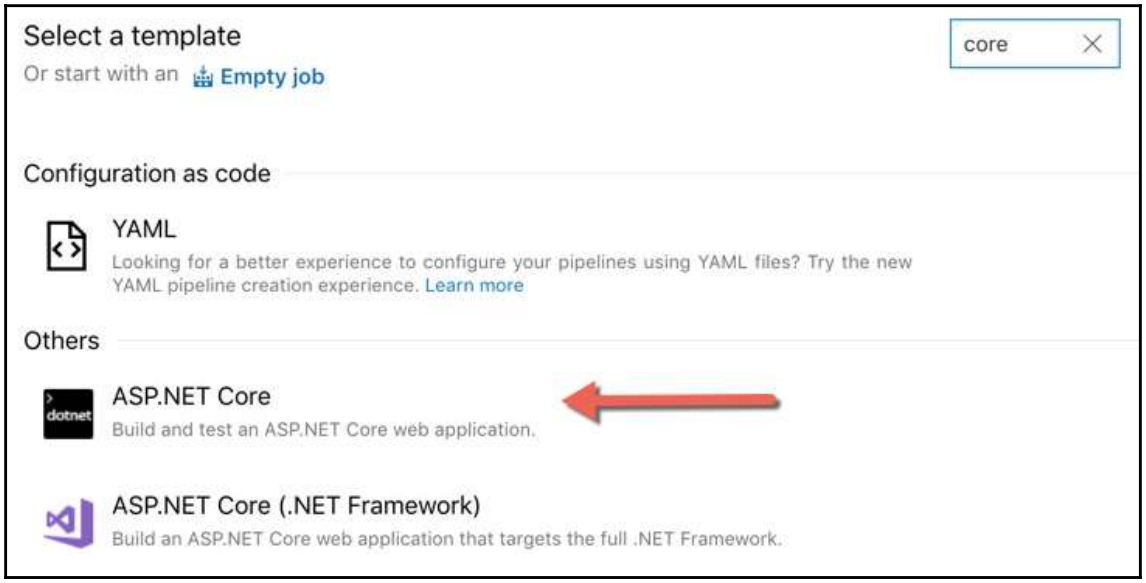
Microsoft introduced .Net Core back in 2016. It has evolved from a framework in preview to a framework that is running business-critical workloads in production. .Net core is an open source, cross-platform, high-performing framework for modern, cloud-based, internet-connected applications. While one had to handcraft build tasks for .Net core applications in its early days, the tooling has now caught up with the pace of change in .Net core. Azure DevOps Server fully supports .Net core and allows you to go from zero to DevOps in a few clicks. In this recipe, we'll learn how to set up a build pipeline for a .NET Core application that can build, unit test, and package the output as an artifact.

Getting ready

In this recipe, we'll be using a simple .Net core application that comprises a few unit tests. To get started, simply import the .Net core sample GitHub repository `https://github.com/MicrosoftDocs/pipelines-dotnet-core` into the parts unlimited team project.

How to do it...

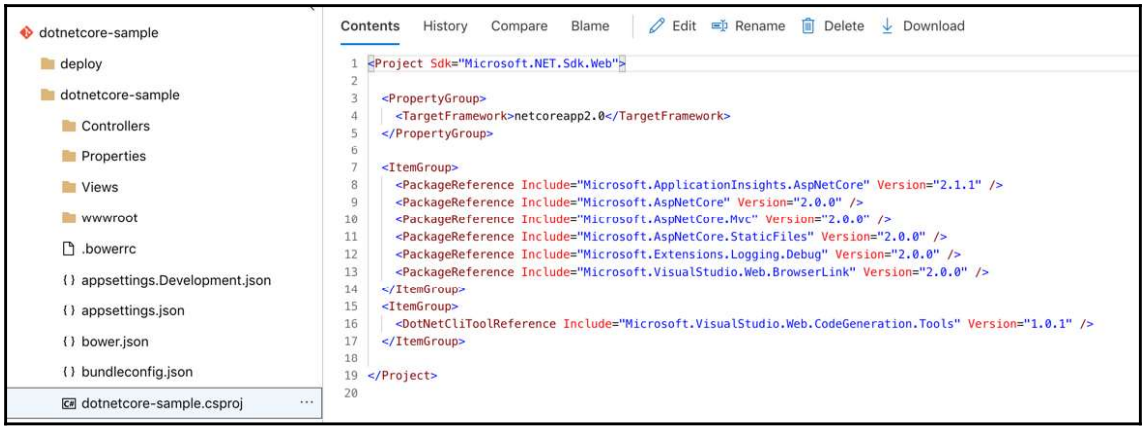
1. Navigate to the build view in the parts unlimited team project. Click **+ New** to create a new build definition and apply the **ASP.NET Core** template:



2. Configure the agent queue to use the default queue and the **Get sources** step to the code repository you've imported the .Net core sample repository into.

How it works...

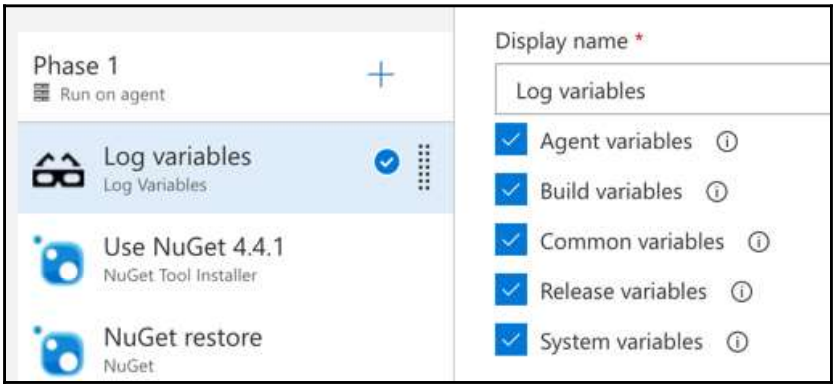
Let's double-click the build process to understand the inner workings of the pipeline better. Start with the restore step. This simply restores all the package dependencies specified in the `csproj` file:



You can also restore package references from NuGet. Use an accompanying `NuGet.config` file in your repository to manage the references to internal or public NuGet feed. More information on how to set this up can be found on Microsoft docs: `https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-restore?tabs=netcore2x`. Version 2 of the VSTS .Net core restore task supports specifying the NuGet feed configuration in the task directly.

There's more...

The Azure DevOps Server marketplace features the **Diagnostics Tasks**, which can be found here: `https://marketplace.visualstudio.com/items?itemName=andremarques023.DiagnosticTasks`. This free extension, developed by André Marques de Araújo, provides you with a set of useful tasks for both build and release pipelines. The log variables task is extremely useful, especially when you are working through debugging build issues. Team build brings a number of predefined variables that can be used in build (and release) definitions and scripts. Variables are generated by the agent in the scope of a particular job (prior to it starting) or generated on the server side and sent to the agent as part of the job. This task logs these variables to the console:

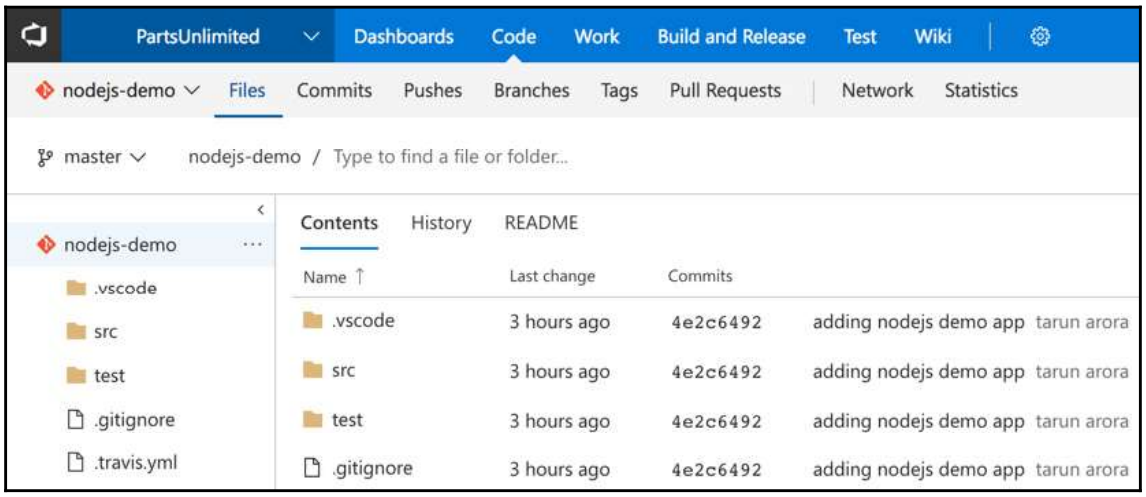


Setting up build pipeline for a Node.js application

Node.js is a cross-platform, open source platform built on Chrome's JavaScript runtime for fast and scalable server-side and networking applications. It is very popular for both frontend as well as server-side programming. In this recipe, you'll learn how to set up a CI pipeline for a Node.js application using gulp.

Getting ready

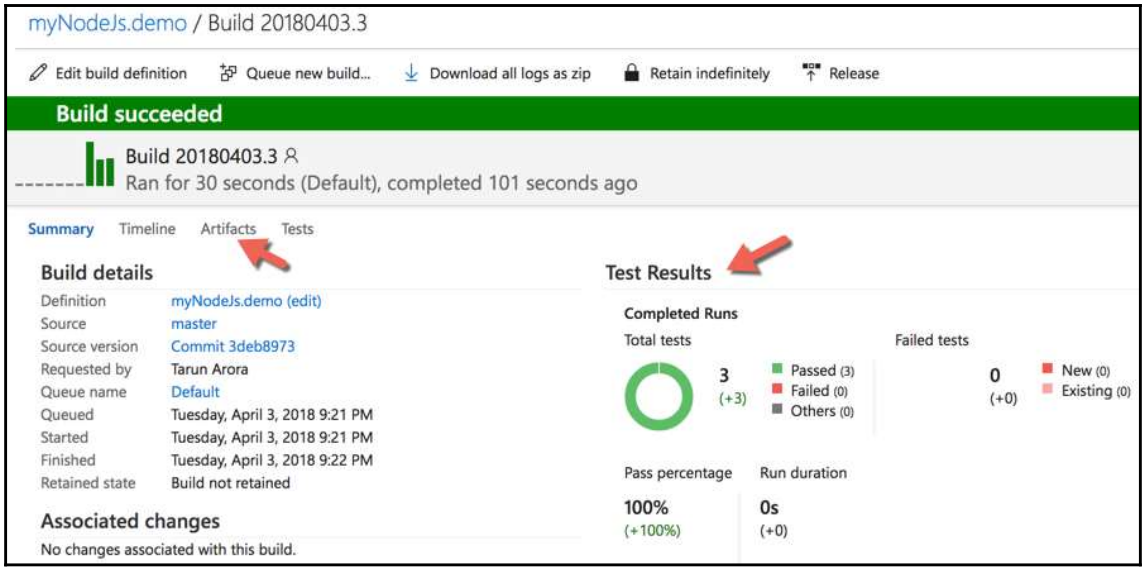
The focus of the recipe is to help you understand the construction of a CI pipeline for a Node.js application. To abstract the complexity of the node application out of the recipe, we'll be using a demo code repository from GitHub. To get started, simply import the following code base from `https://github.com/nilaydshah/MochaTypescriptTest-101/` into your team foundation server. You can also create a new Node.js code repository in Visual Studio code using the instructions in this blog post: `https://blogs.msdn.microsoft.com/nilayshah/2018/01/07/unit-testing-node-application-with-typescript-in-vs-code-%E2%80%8A-%E2%80%8A-using-mocha-chai-mochawesome-gulp-travis/`:



How to do it...

1. Navigate to the build view in the parts unlimited team project. Click **+ New** to create a new build definition and apply the **Node.js With gulp** template:

9. Queue a new build to see your Node.js continuous integration pipeline in action. The pipeline will build, run tests, publish test results, and package the output into a ZIP file that will be attached as an artifact with the build:



How it works...

The `package.json` file is the glue in the Node.js build pipeline. Let's double-click in the build pipeline to understand how this is working under the hood. The first task in the `npm install` pipeline reads `package.json` to identify the application dependencies and restores the packages into the build environment. In the package management chapter, we'll also learn how to plug in a private NPM feed to the build pipeline to restore the package dependencies:

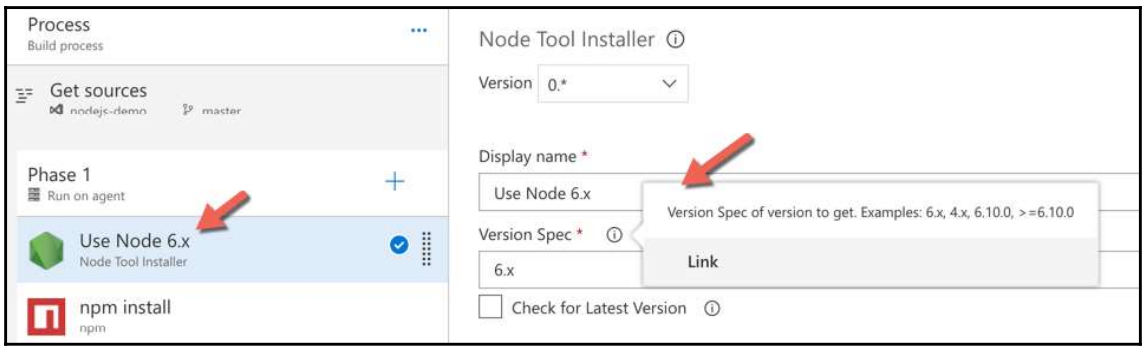
The `package.json` file also includes the custom script to test the application. This calls the mocha test framework and specifies the test output format as `TEST-RESULT.xml`. The output file generated here is compatible with the open test result syntax supported by the build system:

```
"scripts": {
  "test": "mocha lib/test/**/*.js --reporter mocha-junit-reporter --reporter-options mochaFile=./TestResults/TEST-RESULT.xml"
},
```

This allows the build systems to consume the `TEST-RESULT.xml` file through the publish test result task and process it to render the test results visually as part of the build output. Finally, the archive task takes the output processed through the gulp task executor and packages it up into a ZIP file, which is then published as an artifact into the build.

There's more...

You can optionally use the **Node Tool Installer** task to configure the version of Node used by your build pipeline. This task allows you to specify the configuration of the node version to be used in the build pipeline; it accepts the less than, equal to, and greater than expressions. The task finds or downloads and caches the specified version of Node.js and adds it to the path on the build agent host machine:



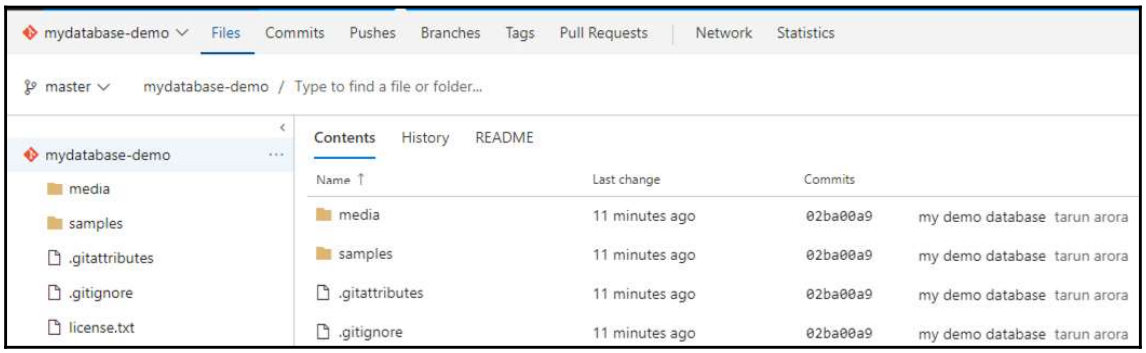
Setting up a build pipeline for your database projects

A continuous integration pipeline ensures code and related resources are integrated regularly and tested by an automated build system. CI is becoming a standard in modern software development. While teams are quick to set up a CI pipeline for their application, the database usually gets sidelined in this equation. The benefits of CI can be applied to brownfield as well as greenfield databases. In this recipe, we'll learn how to set up a pipeline for a database project that generates a `dacpac` file as a build artifact.

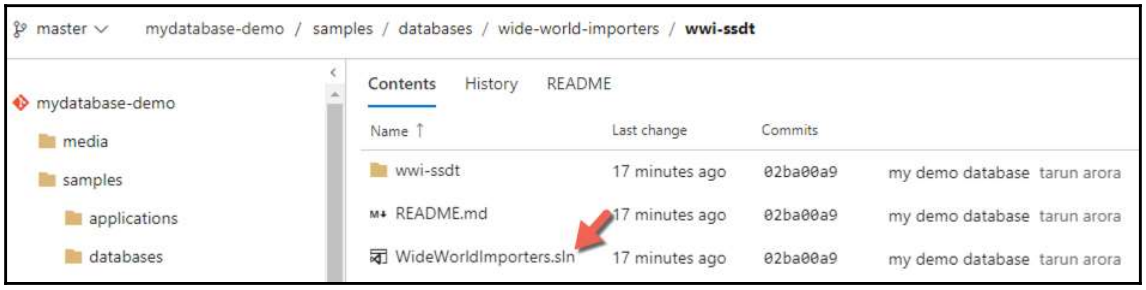
A DAC is a self-contained unit of SQL Server database deployment that enables data-tier developers and database administrators to package SQL Server objects into a portable artifact called a DAC package, also known as a **DACPAC**.

Getting ready

The focus of the recipe is to help you understand the construction of a pipeline for a database project. If you don't have a database project for your database already, you can generate a database project from an existing database using the steps listed here: `https://msdn.microsoft.com/en-us/library/hh864423(v=vs.103).aspx`. In this recipe, we'll be using a simple demo database project from GitHub: `https://github.com/Microsoft/sql-server-samples.git`. To get started, simply import the GitHub repository into the parts unlimited team project:

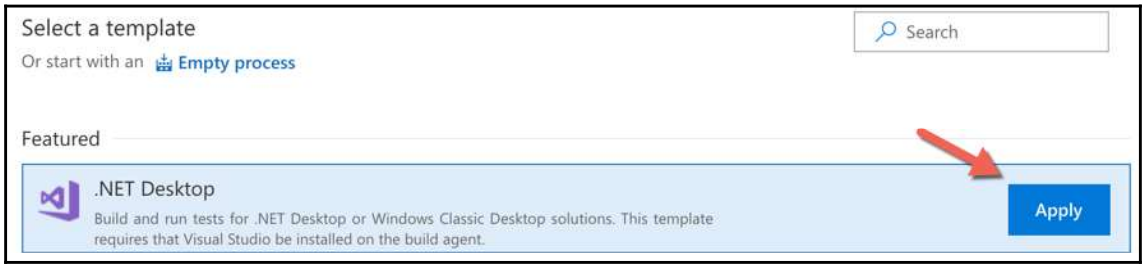


We'll be using the `wwi-ssdt` solution, which already includes the `WideWorldImporters.sqlproj` database project sample:



How to do it...

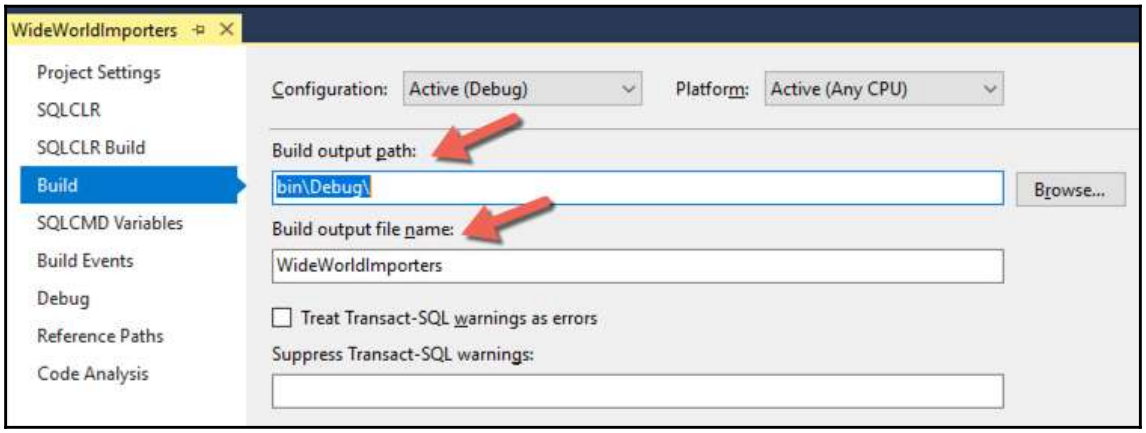
1. Navigate to the build view in the parts unlimited team project. Click **+ New** to create a new build definition and apply the .NET Desktop build template:



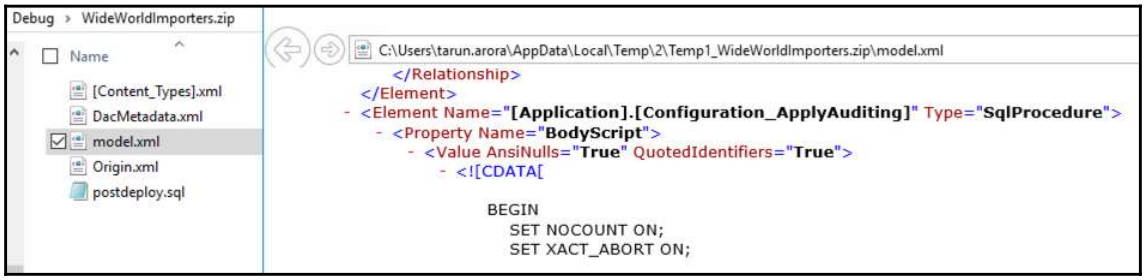
2. Configure the agent queue to use the default queue and, in the process, update the path of the solution file to `samples/databases/wide-world-importers/wwi-dw-ssdt/WideWorldImportersDW.sln`.

How it works...

Open the `samples/databases/wide-world-importers/wwi-dw-ssdt/WideWorldImportersDW.sln` solution in visual studio, right-click on the `WideWorldImporters.dbproj` file, and view properties. In the **Build** tab you'll see that the project is configured to generate an output (`dacpac`) in the `bin\Release` or `bin\Debug` folder:



The build pipeline uses the same setting to generate the `dacpac` file. Download the `dacpac` file generated in the build artifact and rename its extension from `.dacpac` to `.zip`. You'll notice that it simply contains the database model wrapped up into an XML file and the post-deployment scripts in a `postdeployment.sql` file:



The `dacpac` file can be used at deploy time to compare the database against the current state of the schema to generate the incremental delta script for deployment. Refer to *Deploying the database to Azure SQL using the release pipeline* recipe in `Chapter 6`, *Continuous Deployments*, to learn how to deploy dacpac to a sql azure database using Azure Pipelines.

Integrating SonarQube in build pipelines to manage technical debt

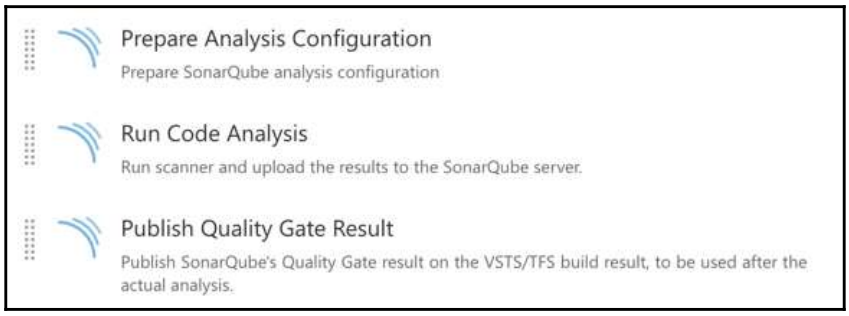
Technical debt can be classified as the measure between the codebase's current state and an optimal state. Technical debt saps productivity by making code hard to understand, easy to break, and difficult to validate, in turn creating unplanned work, ultimately blocking progress. Technical debt is inevitable! It starts small and grows over time through rushed changes, lack of context, and lack of discipline. Organizations often find that more than 50% of their capacity is sapped by technical debt. The hardest part of fixing technical debt is knowing where to start. SonarQube is an open source platform that is the de facto solution for understanding and managing technical debt. In this recipe, we'll learn how to leverage SonarQube in a build pipeline to identify technical debt.

Getting ready

SonarQube is an open platform to manage code quality. As such, it covers the seven axes of code quality as illustrated in the following diagram. Originally famous in the Java community, SonarQube now supports over 20 programming languages. The joint investments made by Microsoft and SonarSource make SonarQube easier to integrate with TFBuild and better at analyzing .NET-based applications. You can read more about the capabilities offered by SonarQube here: `http://www.sonarqube.org/resources/`

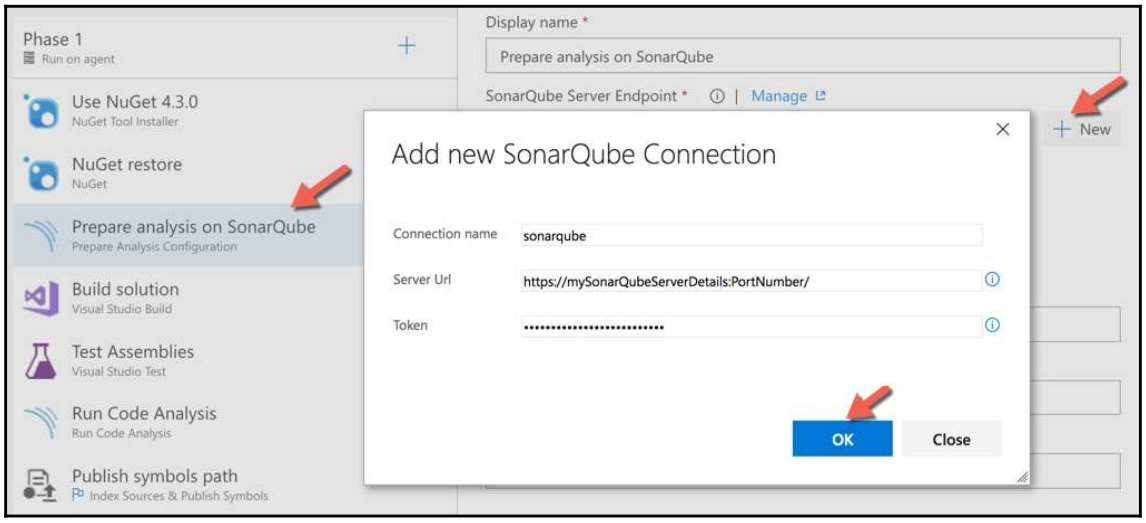
In this recipe, we'll be analyzing the technical debt in one of the .Net core sample repositories in the `partsunlimited` team project. If you don't already have an instance of SonarQube, then set one up by following the instructions here: `https://github.com/SonarSource/sonar-.net-documentation/blob/master/doc/installation-and-configuration.md`.

To get started with SonarQube, you'll also need to install the SonarQube build tasks to your Azure DevOps Server Team Project collection from the marketplace: `https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarqube`:

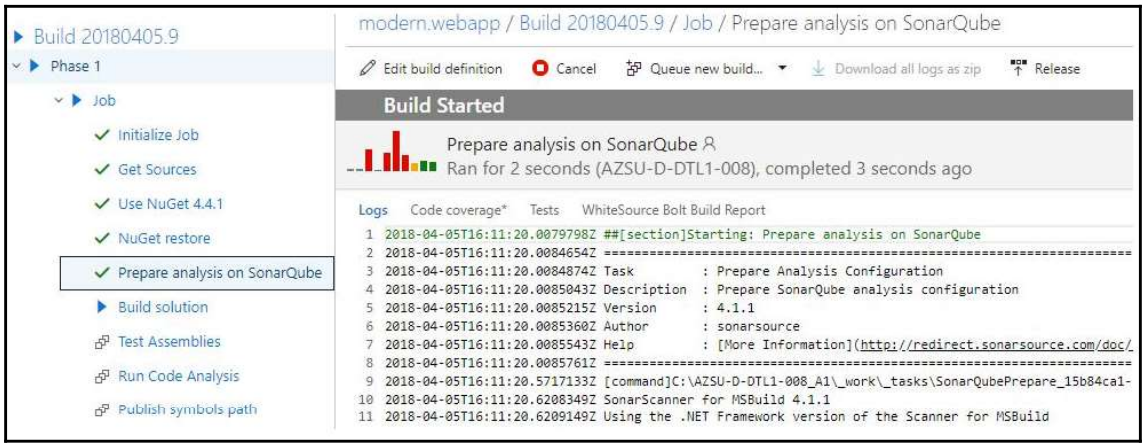


How to do it...

1. Navigate to the build view in the parts unlimited team project.
2. Choose to edit the `modern.webapp` build definition, click **+**, and add the following tasks: **Prepare analysis on SonarQube** and **Run Code Analysis**.
3. Click on the **Prepare analysis on SonarQube** task and click **+ New** to configure the SonarQube service endpoint to be used:



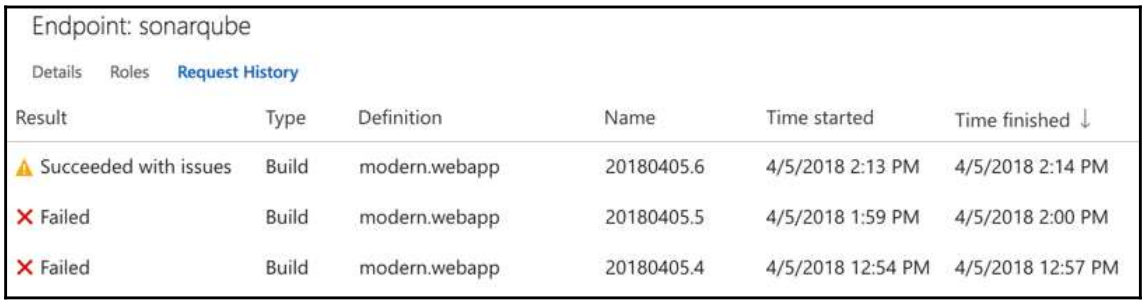
5. Queue a new build and wait for the build execution to complete:



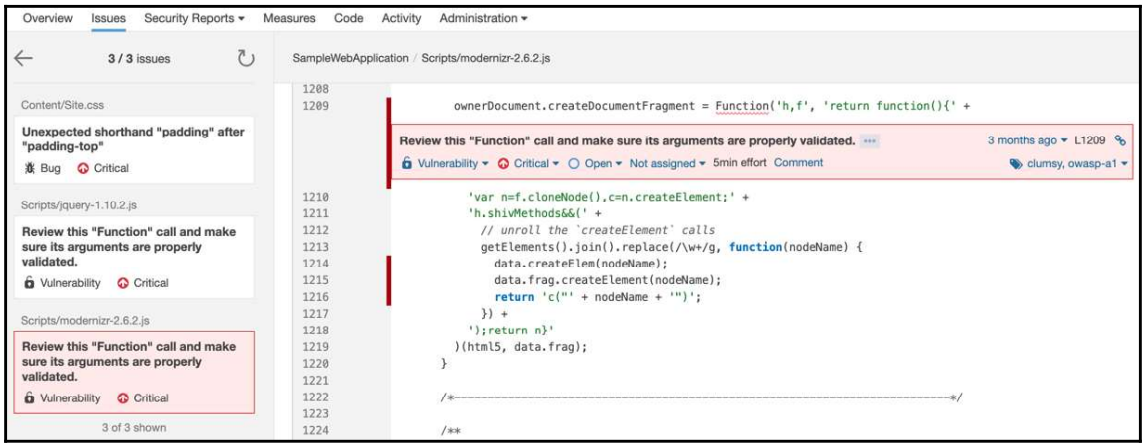
The build has successfully run, completed the SonarQube analysis, and pushed the results into your SonarQube instance.

How it works...

The build tasks provided by SonarQube provide the underlying plumbing to leverage the correct analyzers, generate the analysis report, and publish it to the SonarQube instance specified in the build pipeline. The service endpoint created for SonarQube keeps track of all the requests that make use of this service endpoint:



The analysis shows that there are three major issues; click on the issues to see more details about them. This view gives you the option to slice and dice the issues by various categories. It's possible to click on the issue and see the offending line of code with details of how this can be fixed:



The measures in SonarQube give you the ability to get a better all-around view of the quality of your application. For example, in the duplication measure it is demonstrated that the application is plagued by 17% of code that can be refactored to more shared functions, as in its current state it's simply duplicated:

There's more...

You can optionally edit the pipeline to include the **Publish Quality Gate Results** task in the pipeline. This task publishes a summary of the SonarQube code analysis results into the build summary view.