5
Continuous Testing

Software teams are constantly under pressure to deliver more, faster. End users expect software to simply work. Low-quality software just isn't acceptable. But you may ask what the right level of quality is. Quality is a very subjective term; it is, therefore, important for teams to agree on a definition of quality for their software. Teams that are unable to define quality usually end up testing for coverage.

Microsoft has made some bold bets with Azure DevOps Server 2019. Rather than continue to invest in features that have a very high cost of ownership and low usability footprint, Microsoft has instead decided to deprecate those features and instead focus the energy elsewhere. Let's see what's changing:

- **Microsoft Test Manager (MTM)**: The toolkit in Azure DevOps Server provides tooling for both manual and automated testing. A key part of that tooling used to be MTM. MTM was first introduced with TFS 2010. It enabled testers to plan, track, and run manual tests, exploratory tests, and automated tests. While MTM fully integrated with TFS, it did not offer integration with other testing platforms, nor did it offer APIs for extensibility. Microsoft's ambition over the last few years has been to support every developer, every app, and every platform; that isn't possible with tooling that can't be run on non-Windows platforms. As a result, over the years, test tooling has gradually moved out of MTM onto the web, which is now called Test Hub. As it stands, Test Hub is a fully-featured test management solution spanning all stages of the testing life cycle. It works on all platforms (such as Linux, macOS, and Windows) and all browsers (such as Edge, Chrome, and Firefox). You can easily get started using manual testing features right from your Kanban board and use it for more advanced manual testing capabilities.

With the deprecation of MTM, load testing, and Coded UI Testing, you are probably thinking: where is Microsoft investing in testing? I'll answer that question, but let's first look at this interesting shift. To speed up the software delivery loop, software testing needs to be incorporated into the continuous integration pipeline. In order to do this, software testing needs to shift left in the development processes. Test-driven development enables developers to write code that's maintainable, flexible, and easily extensible. Code backed by unit tests helps identify change impact and empowers developers to make changes confidently. In addition to this, functional testing needs to be automated. This enables software testers to focus on high-value exploratory testing rather than just coverage of the test matrix. The DevOps movement at large supports bringing testing into the continuous integration pipeline. As a result, the next wave of investment is going into improving the testing story within pipelines, specifically around unit testing: more support for testing frameworks and enriching the analytics from test execution.

Through the recipes in this chapter, we'll learn how to leverage pipelines to execute tests and perform distributed test execution.

In this chapter, we will cover the following recipes:

- Running NUnit tests using Azure Pipelines
- Using feature flags to test in production
- Distributing multi-configuration tests against agents
- Configuring parallel execution of tests using Azure Pipelines
- Running SpecFlow tests using Azure Pipelines
- Analyzing test execution results from Runs view
- Exporting test artifacts and test results from Test Hub
- Charting testing status on dashboards in the team portal

Running NUnit tests using Azure Pipelines

NUnit is one of the many open source testing frameworks popular with cross-platform developers. In this recipe, we'll learn how easy it is to create a pipeline for NUnit-based tests and publish the test execution results in Azure DevOps Server.

Getting ready

In this section, we'll use the .NET CLI to create a new solution and a new class library project, and install the NUnit test template.

The `[TestFixture]` attribute denotes a class that contains unit tests. The `[Test]` attribute indicates that a method is a test method. Save this file. From the command line, execute `dotnet test`; this builds the tests and the class library and then executes the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
public bool IsPrime(int candidate)
{
 if (candidate == 1)
 {
 return false;
 }
 throw new NotImplementedException("Please create a test first");
}
```

In the `ContinuousTesting` directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

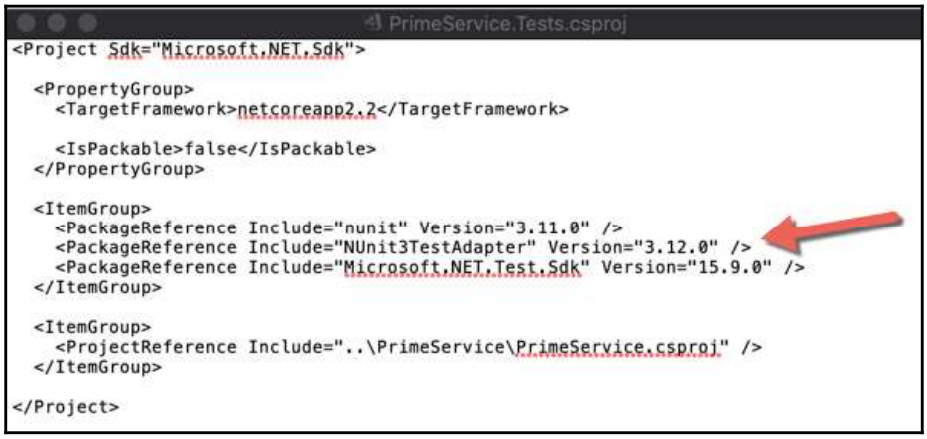
Now that we have a working .Net core service and NUnit-based unit test, commit the code into a Git repository (be sure to use a `gitignore` file to avoid staging files you don't need). Create a remote `continuoustesting.demo` repository in the **Parts Unlimited** team project. Push the code into the master branch on the remote.

How to do it...

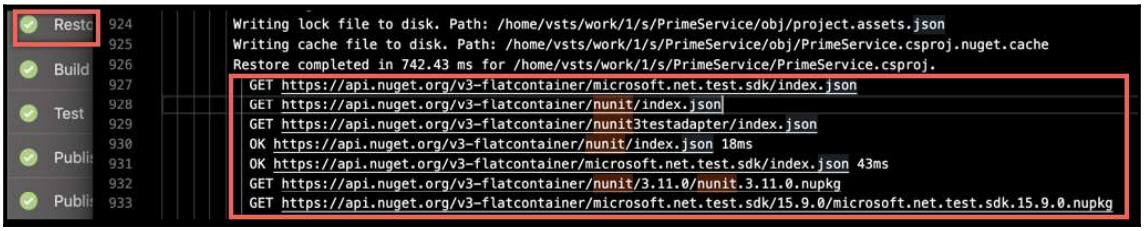
1. Navigate to the **Build** view in the Parts Unlimited team project.
2. Click **+New** to create a new pipeline and apply the `dotnetcore` template. Name the pipeline `nunit.demo`.

How it works...

This was simple! You didn't have to add any reference to the NUnit test runner in the pipeline or worry about parsing the NUnit test results back into a format that is understood by the pipeline. The Azure DevOps service does a lot of work behind the scenes to make it seamless. To understand how it works, let's start by zooming into the **Restore** step in the pipeline. The pipeline reads the `csproj` reference to the NUnit test adapter:



As a result, the test runner is downloaded:



Azure Pipelines are highly extensible and provide a wide range of extensibility points. The test task out-of-the-box supports the following test result formats: CTest, JUnit, NUnit 2, NUnit 3, Visual Studio Test (TRX), and xUnit 2. The test task is executed through the pipeline supports parsing the test results from any test execution framework as long as the test framework can publish the test results in any of these supported formats. All the advanced concepts of searching test results using wildcard search as well as merging test results are handled by the pipeline itself.

Using feature flags to test in production

We are in an era of continuous delivery, where we are expected to quickly deliver software that is stable and performant. We see development teams embracing a suite of continuous integration/delivery tools to automate their testing and QA, all while deploying at an accelerated cadence. No matter how hard we try to mitigate the risk of software delivery, almost all end-user software releases are strictly coupled with some form of code deployment. This means that companies must rely on testing and QA to identify all issues before a release hits production. There are two key challenges when testing features in test environments:

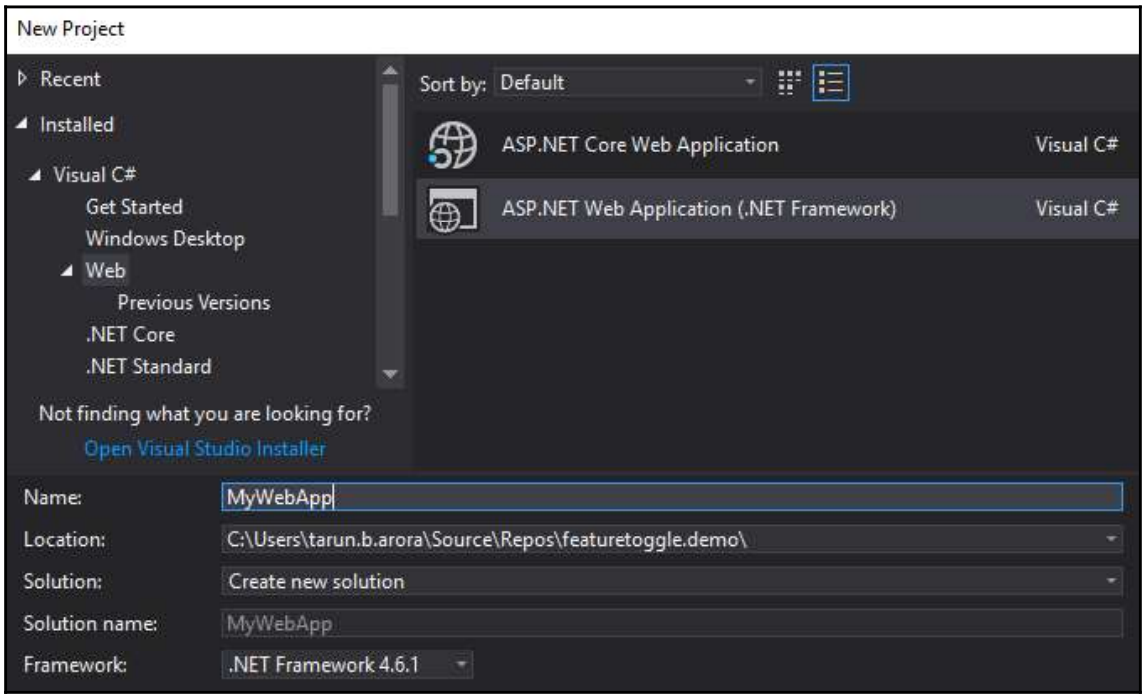
- Testing in test environments can be challenging if your test scenarios depend on production-quality data. It can take a lot of effort to create this kind of data in test environments and it's likely you'll still miss out on key test scenarios, since in some cases the effort involved in creating this data outweighs the benefits.
- The other most common scenario is doing user testing, inspecting, and adapting the functionality of your product based on usage data. End users may be invested in the success of your product, but it can get increasingly difficult to get constant feedback on every functionality in a test environment.

Once a release is in production, it is basically out in the wild. Without proper controls, rolling back to previous versions becomes a code deployment exercise, requiring engineering expertise and increasing the potential for downtime. One way to mitigate risk in feature releases is to introduce feature flags (feature toggles) into the continuous delivery process. These flags allow features (or any code segment) to be turned on or off for particular users. Feature flags are a powerful technique, allowing teams to modify system behavior without changing code. Innovation is the key to success, and success depends on hypothesis testing through experimentation. By adopting a culture of continuous experimentation, features can be tested by creating an instrumented minimal viable product rapidly and released to a subset of customers in production for testing; this enables the team to make fact-based decisions and quickly evolve toward an optimal solution.

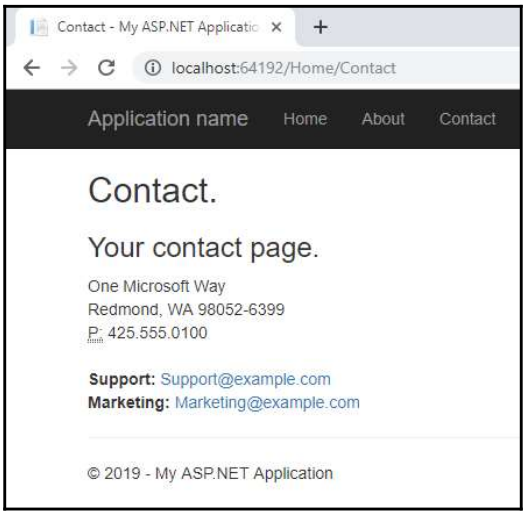
In this recipe, we'll learn how to get into a true continuous testing culture by leveraging feature flags.

Getting ready

1. Create a new web application using the ASP.NET Web Application template in Visual Studio, name it `MyWebApp`, and save it in a new folder called `featuretoggle.demo`:



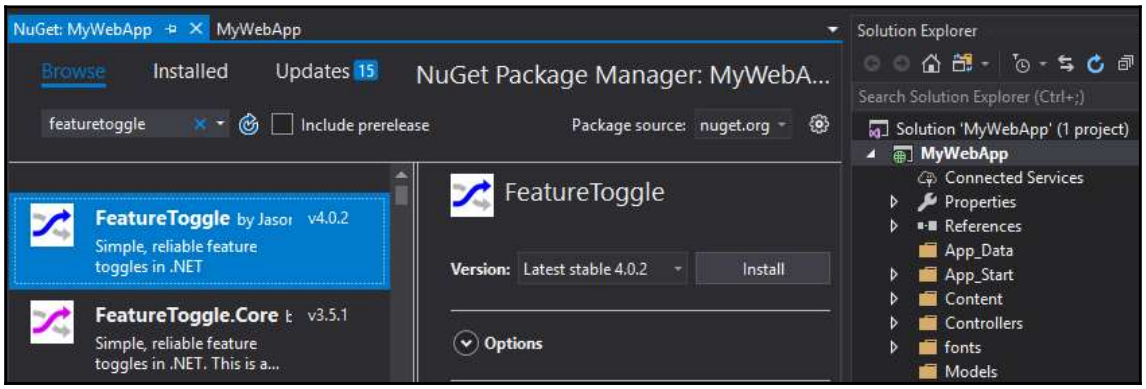
2. Simply build and run the website, then navigate to the **Contact** form:



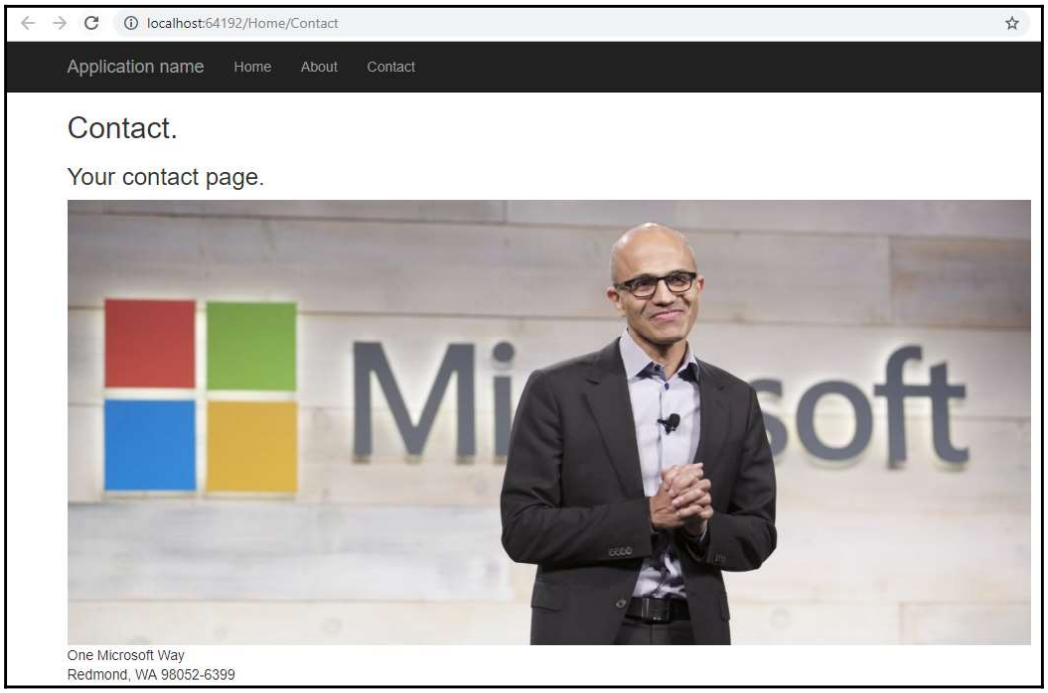
In the next section, we'll see how to use feature flags to deploy changes to the **Contact** form without releasing the changes to everyone.

How to do it...

1. In the `MyWebApp` project, add a reference to the `FeatureToggle` package:



6. Now refresh the **Contact** form. You'll see the updated page with the image:



How it works...

The feature toggle package includes a series of providers that can be used to control the value of an object that can, in turn, be used to decide whether the feature is accessible. You may ask why we use feature toggle. Well, it is easy to construct a simple `if....else` condition using a config key to control when the page gets shown. While magic strings can be used, toggles should be real things (objects), not just a loosely typed string. This helps effectively manage the feature flags over time. When using real toggles, you can do the following:

- Find uses of the `Toggle` class to see where it's used
- Delete the `Toggle` class and see where a build fails

Feature flags allow you to decouple code deployments from feature releases. This simplifies testing code changes in production without impacting end users. By using feature flags, it's possible to control who can see a feature; it's also possible to phase in traffic to a new feature rather than opening up all users at once. You can read more about feature flags and their benefits here: `https://martinfowler.com/articles/feature-toggles.html`.

There's more...

The feature toggle package also provides the following feature toggle types:

- `AlwaysOffFeatureToggle`
- `AlwaysOnFeatureToggle`
- `EnabledOnOrAfterDateFeatureToggle`
- `EnabledOnOrBeforeDateFeatureToggle`
- `EnabledBetweenDatesFeatureToggle`
- `SimpleFeatureToggle`
- `RandomFeatureToggle`
- `EnabledOnDaysOfWeekFeatureToggle`
- `SqlFeatureToggle`
- `EnabledOnOrAfterAssemblyVersionWhereToggleIsDefinedToggle`

More details on these feature toggle types and their usage can be found at: `http://jason-roberts.github.io/FeatureToggle.Docs/pages/usage.html`.

Distributing multi-configuration tests against agents

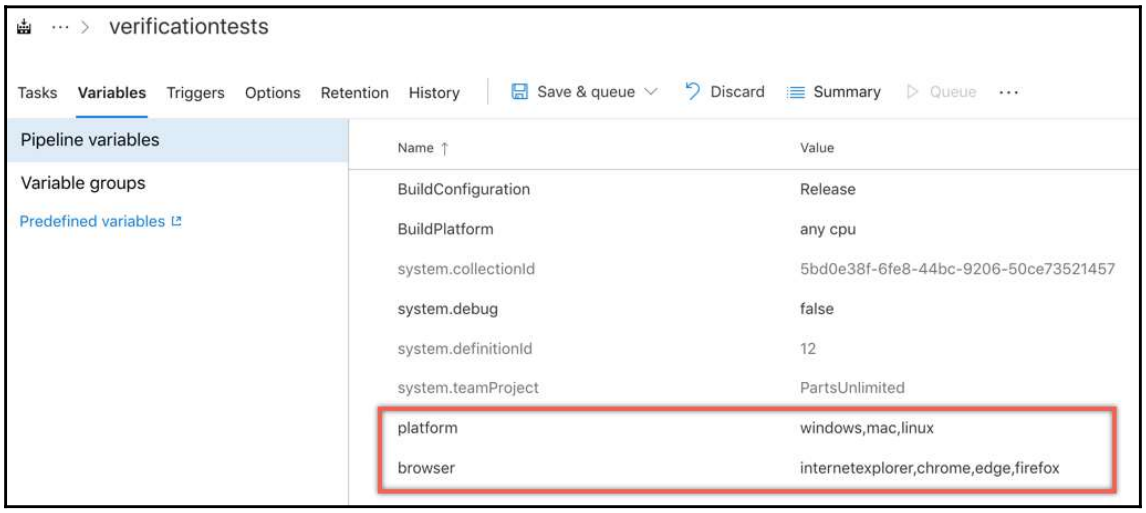
Pipelines are a great way of running tests. The pipeline can be used to run unit tests, functional tests, and integration tests. If you have a large number of tests in your application, the verification process can slow down significantly. It can get even slower if you have a large matrix of configurations to run the tests against. For example, if you have a collection of selenium tests that perform UI-level verification, you may need to run these tests against Internet Explorer, Chrome, and Firefox and run the tests on Windows, macOS, and flavors of Linux.

In this recipe, we'll learn how easy it is to use a combination of a multi-configuration execution plan along with a pool of test agents to distribute the test execution.

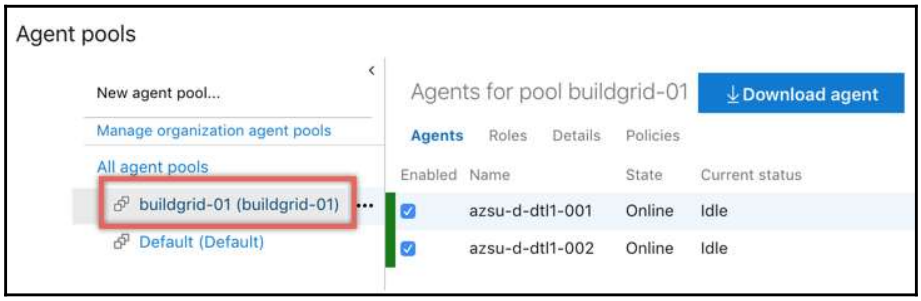
How to do it...

In the **Variables** section in a build pipeline, define one or more variables that'll be used to describe the test matrix:

1. In our example, we need to test against multiple browsers on multiple platforms. So, I've created two variables, one for browsers and the other for platforms:



2. Next, create an agent pool with multiple agents. For the purposes of this recipe, I've created a build pool `buildgrid-01` with two agents:

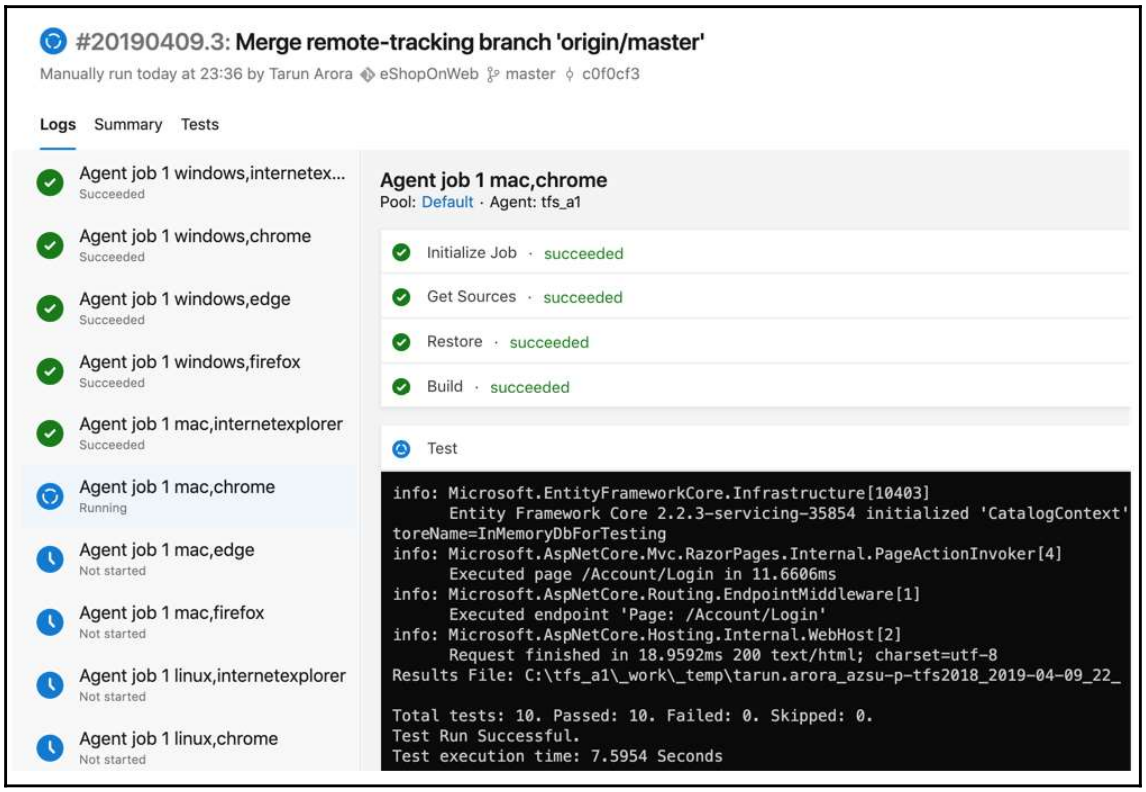


How it works...

The comma-separated values in the `platform` and `browser` variables are used to create the test matrix:

Browser	Internet Explorer	Chrome	Edge	Firefox
Platform	Windows	Windows	Windows	Windows
Platform	Mac	Mac	Mac	Mac
Platform	Linux	Linux	Linux	Linux

The multi-configuration test execution plan simply iterates through the comma-separated values one at a time and passes them to the `$platform` and `$browser` variables, which are then passed in to the test configuration. As you can see in the following screenshot, the test configuration is distributed across the two agents available in the pool:



There's more...

With multi-configuration you can run multiple jobs, each with a different value for one or more variables (multipliers). If you want to run the same job on multiple agents, then you can use the multi-agent option of parallelism. The preceding test slicing example can be accomplished through the multi-agent option.

Configuring parallel execution of tests using Azure Pipelines

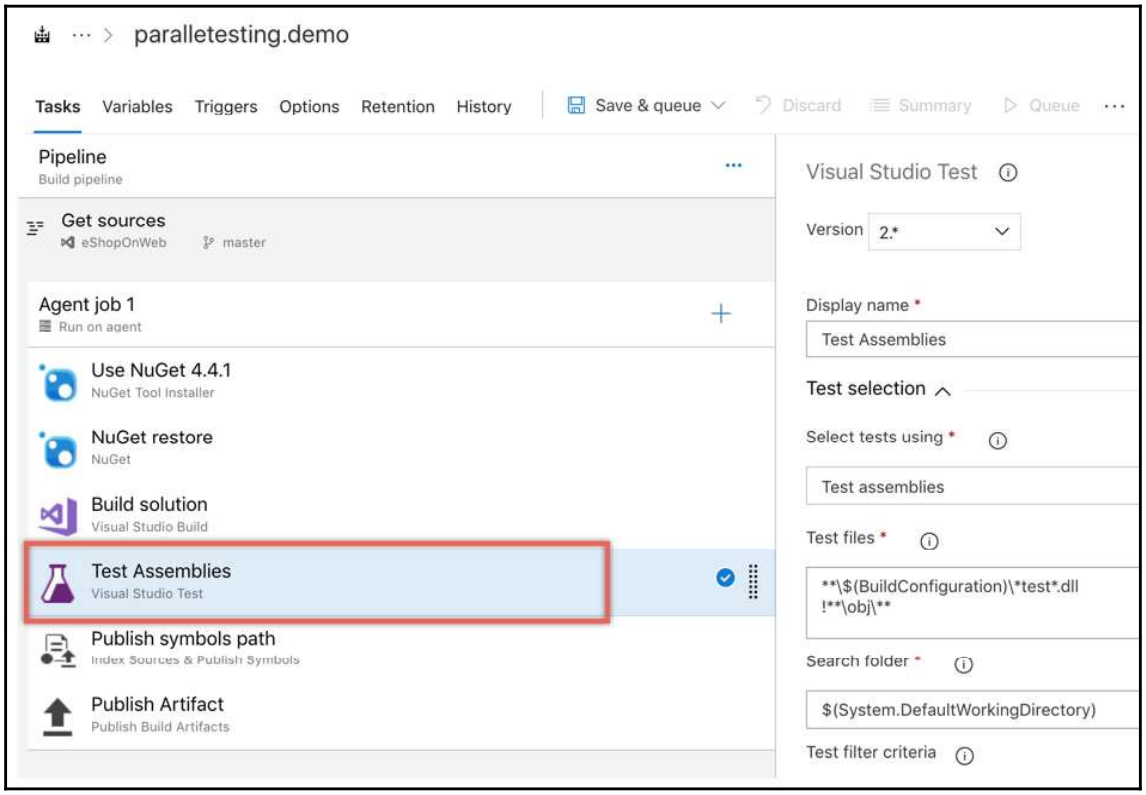
Running tests to validate changes to the code is key to maintaining quality. For continuous integration practice to be successful, it is essential you have a good test suite that is run with every build. However, as the code base grows, the regression test suite tends to grow as well, and running a full regression test can take a long time. Sometimes, tests themselves may be long-running – this is typically the case if you write end-to-end tests. This reduces the speed with which customer value can be delivered, as pipelines cannot process builds quickly enough.

Being able to divide the test execution on multiple cores across a pool of agents can significantly reduce the time it takes to complete the test execution. While most build servers are multi-core, the agent orchestrating the pipelines doesn't always provide an easy way to distribute the test execution on multiple cores. In this recipe, we'll see how easy it is to enable parallel execution of tests using Azure Pipelines.

Getting ready

Create a new pipeline using the ASP.NET Core template. This will add the Visual Studio Test task to the pipeline.

Save the pipeline as `paralleltesting.demo`:

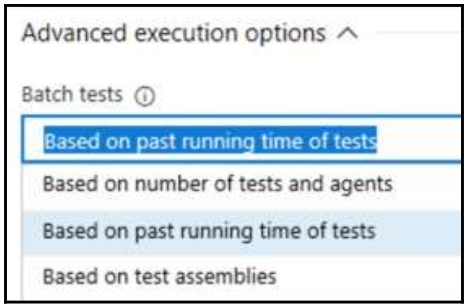


How to do it...

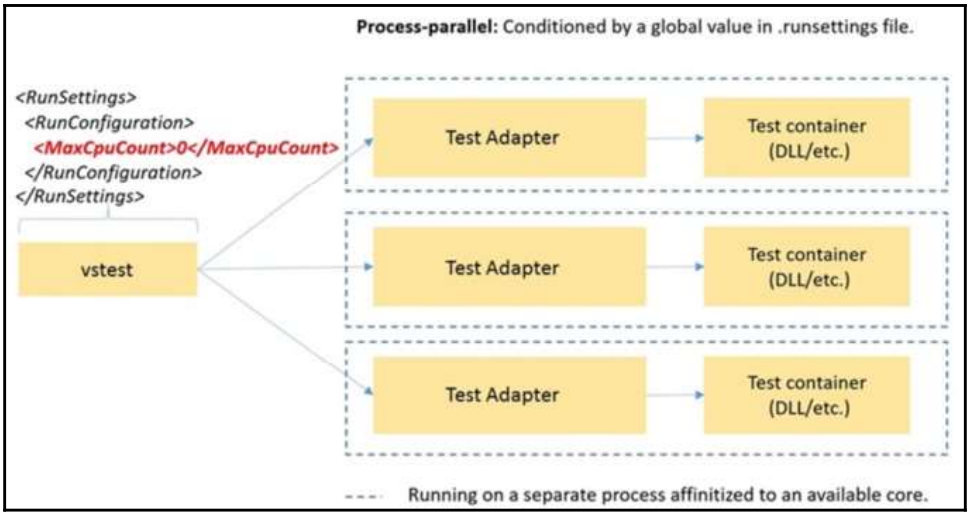
1. In the **Execution** section of the Visual Studio Test task, check the option to **Run tests in parallel on multi-core machines**:

How it works...

The Visual Studio Test task (version 2) is designed to work seamlessly with parallel job settings. When a pipeline job that contains the Visual Studio Test task is configured to run on multiple agents in parallel, it automatically detects that multiple agents are involved and creates test slices that can be run in parallel across these agents. Furthermore, the task can be configured to create test slices to suit different requirements such as batching based on the number of tests and agents, the previous test running times, or the location of tests in assemblies:



When the run parallel checkbox is checked, behind the scenes the `maxcpucount` value is set to 0, which internally configures the Visual Studio Test task to enforce that the test execution process isn't allocated affinity to just one CPU processor:



There's more...

The parallelism of test execution is offered by most test frameworks. All modern test frameworks, such as MSTest v2, NUnit, xUnit, and others, provide the ability to run tests in parallel. Typically, tests in an assembly are run in parallel. The Visual Studio test task already supports the previously listed testing frameworks, therefore the options of parallel execution and slicing based on the number of agents/tests and test assemblies is available to all supported testing frameworks.

Running SpecFlow tests using Azure Pipelines

SpecFlow is a testing framework that lets you define application behavior in plain, meaningful English text using a simple grammar defined by a language called **Gherkin**. SpecFlow is a very popular open source framework for **Behavior-Driven Development** (**BDD**). SpecFlow democratizes testing to non-technical users by giving them a way of defining tests using the business domain and functional language, which can then be fleshed out as a functional test. In this recipe, we'll learn how SpecFlow tests can be integrated to run in Azure Pipelines.

Getting ready

Create a new pipeline using the ASP.NET Core template. In this recipe, we'll be mostly focusing on the Test task in this pipeline.

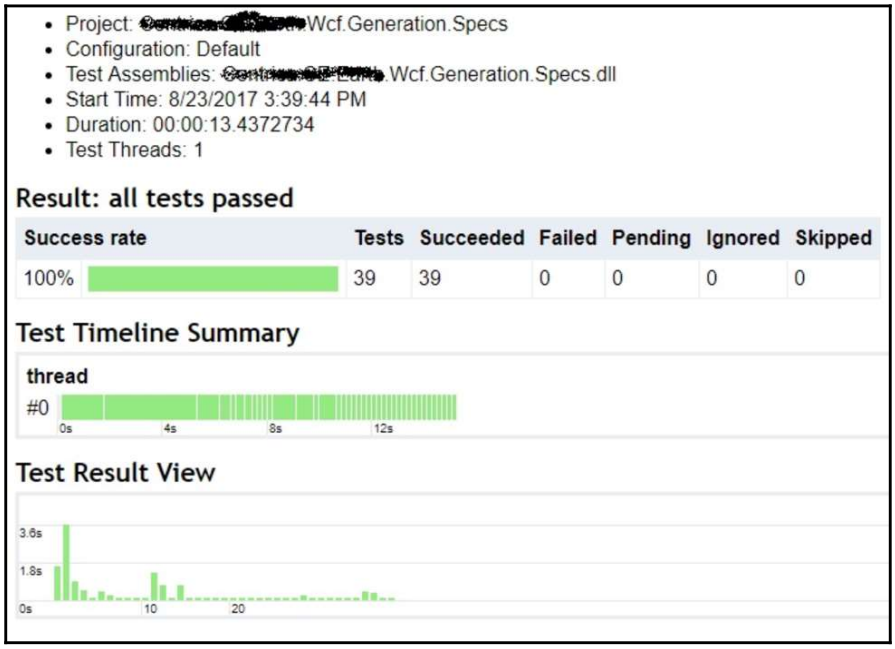
How to do it...

SpecFlow tests don't necessarily need the SpecRunner for execution: they can be run using MSTestv2 or any other compatible framework. However, using SpecRunner provides great benefits: for example, you can get some very useful analysis out of the tests that wouldn't necessarily be available if you used other test execution frameworks. Luckily, using SpecRunner for test execution doesn't require any installation on the agent!

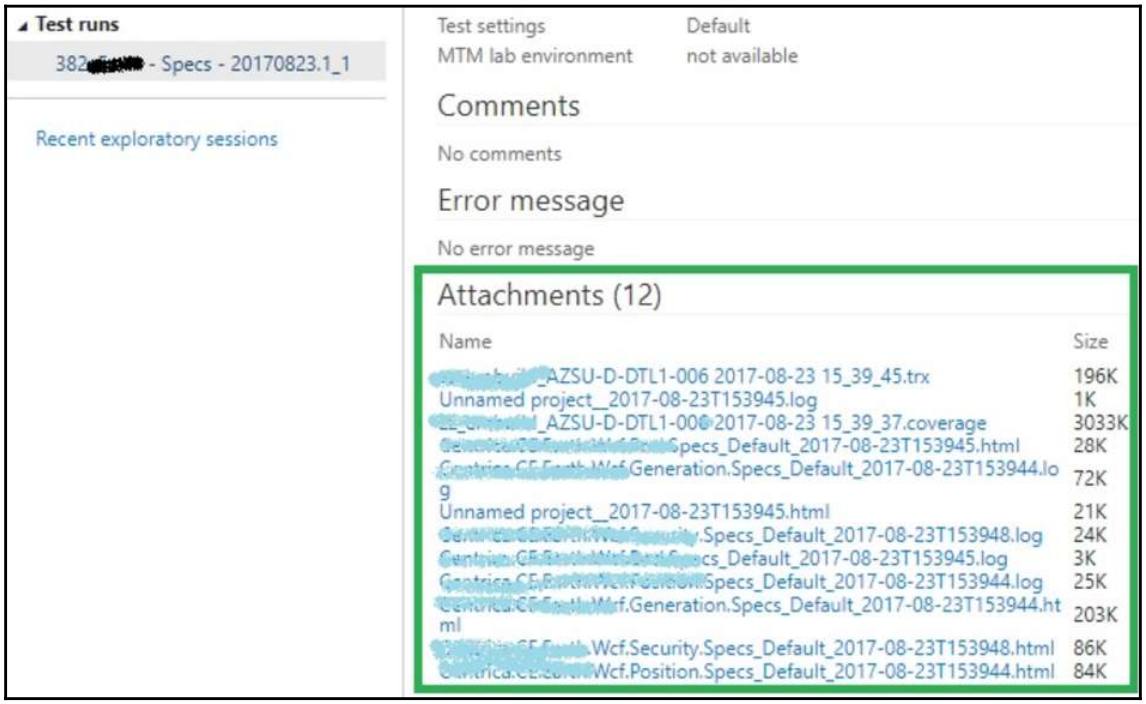
3. If the packages are added correctly, in the **Build** summary for this step you'll be able to see that the tests are executed using the **SpecFlow+ Runner** test adapter:

How it works...

When SpecFlow tests are executed in Visual Studio, an analysis report is generated by SpecFlow:

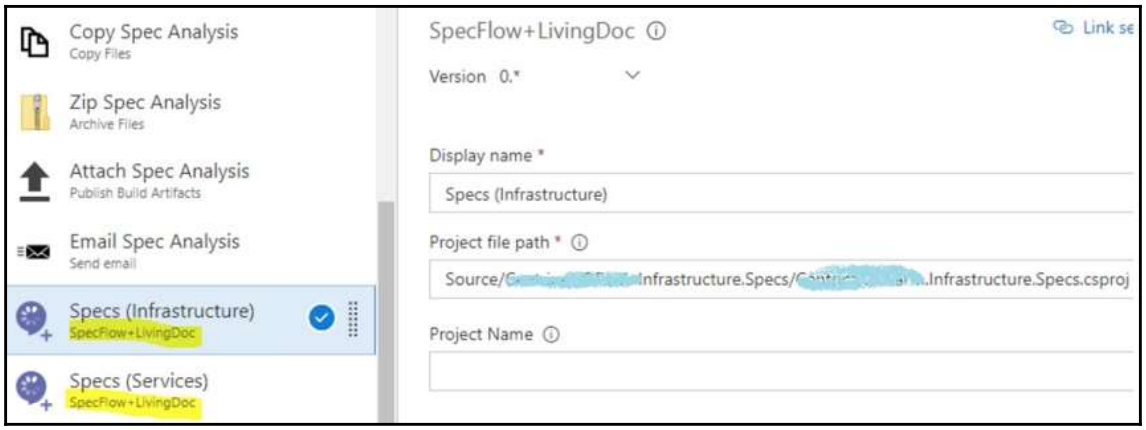


If the **Upload Attachment** option is checked in the test task, the SpecFlow test execution and analysis logs get attached to the test run results:

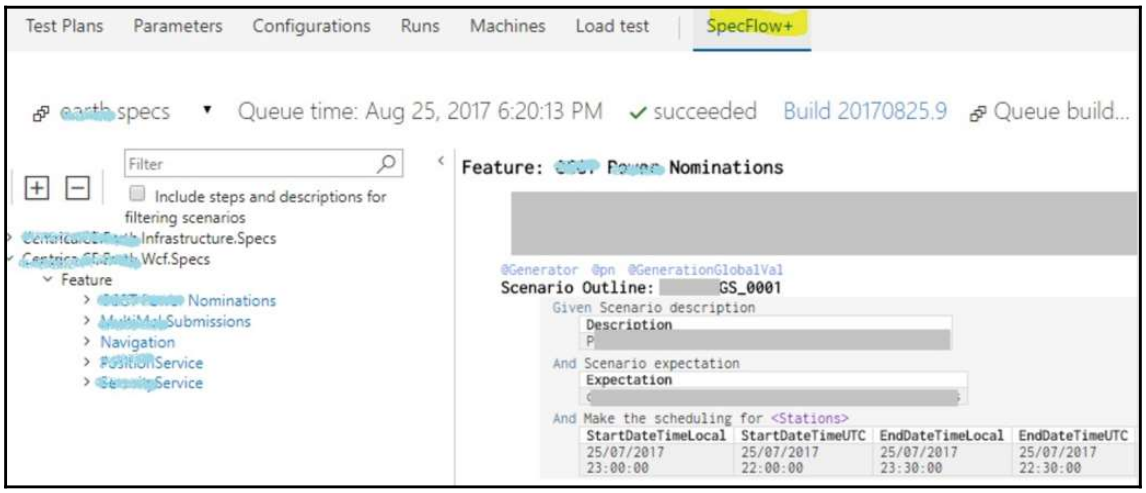


There's more...

By using the SpecFlow plus extension, available in the Azure DevOps Server marketplace you can easily publish your spec tests as living documentation within Azure DevOps Server. This can be achieved by using the SpecFlow+LivingDoc documentation (`https://marketplace.visualstudio.com/items?itemName=techtalk.techtalk-specflow-plus`) extension in your Azure Pipeline:



Personally, I think the SpecFlow+LivingDoc is pretty rough and needs some more work, but nonetheless, it provides great value even in its current state:



Analyzing test execution results from Runs view

In Azure DevOps Server 19, test execution results of both manual and automated testing are surfaced in the Runs page. The Runs page offers a unified experience for analyzing the results of tests executed using any framework. In this recipe, we'll learn how to analyze and action the test execution results in the Runs view in Team Web Portal.

Getting ready

Launch the Parts Unlimited team project, navigate to the **Test Hub**, and click on **Runs** to load the **Runs** page.

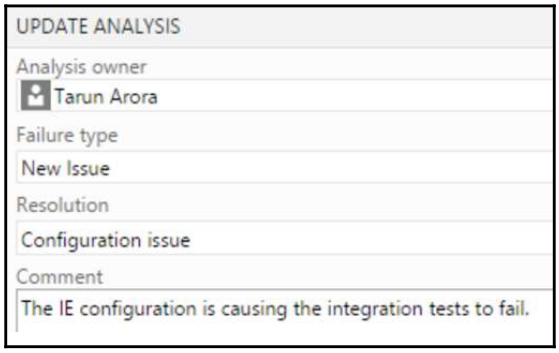
How to do it...

The **Runs** page displays the recent test runs. At first glance, you can see the test execution status, test configuration, build number, number of failed tests, and pass rate:

1. Navigate to the filters view by clicking the **Filters** tab. The query is defaulted to display the test runs from the last seven days.
2. Amend and add new clauses to show only the automated test runs for today:

The query narrows down the test execution results to just one run:

6. Click on **Update analysis** to add comments to the test results. You can also double-click a test to go to the next level of detail on its test execution:



How it works...

This functionality gives you a unified test analysis experience irrespective of the framework on which you choose to execute your tests. In summary, you can query all test runs available in your Team Project, drill down into a specific test run to get a summary view of that run, visualize test runs using charts, query/filter the test results within a run, drill down to a specific test result, download attachments, and, last but not least, analyze test failures and file bugs.

Exporting test artifacts and test results from Test Hub

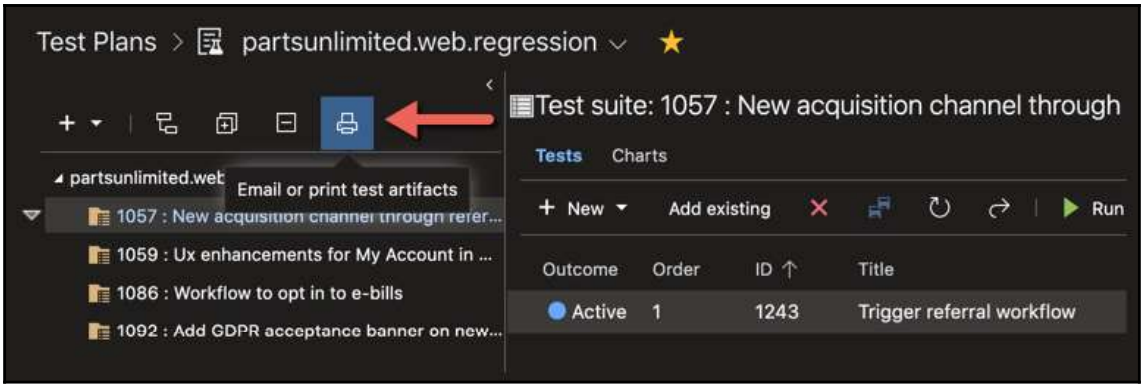
In Azure DevOps Server, test artifacts comprise test plans, test suites, test cases, and test results. It is common to have to export the test artifacts for the purposes of sharing and reporting. Back in the days of TFS 2013, Test Scribe delivered as a Visual Studio Extension was the only way to export these artifacts. Test Hub now boasts the email or print test artifacts functionality, which allows you to easily share test artifacts with stakeholders. The feature is simple to use and can be triggered from several places within the Test Hub.

Getting ready

Launch the Parts Unlimited team project and navigate to the Test Hub.

How to do it...

1. Select the **Test Plans** and click on **Email or print the test artifacts** from the toolbar:



You can export the artifacts from the root by selecting the top-level test suite.

Whether you chose to export from test plan or test suite in both the cases, you will get a new form to select 'what' and 'how', the 'what' in this case being the artifacts, and the how being email or print. A few items are worth highlighting in the following screenshot. The **Latest test outcome** option has been added in Update 1; selecting this option also exports the test results.

2. Choosing **Selected suite + children** recursively exports all children of the selected suite:

How it works...

Clicking on print or email starts the process of generating the extract. This may take up to a few seconds to complete, depending on the quantity and size of the artifacts being exported. Once the export has been completed, a form will pop up to show you the preview of the export. You can also edit and format the values from the preview form.

There's more...

It is possible to customize the format of the export by modifying the underlying template used by Azure DevOps Server during the export/print process. There are a few points to keep in mind before customizing the template.

You should create a backup of the original template; for example, copy it and rename it as `TestSuite-Original.xsl`. If not, when you upgrade Azure DevOps Server, the changes you made in the `TestSuite.xsl` file may get overwritten. The export does not support customization per project and the style changes will affect all projects in your Azure DevOps Server instance.

Follow the steps listed here to add your company logo to the export:

1. Log on to the Team Foundation Server application tier and navigate to the following path and add your company logo (`companylogo.png`) in this folder path: `C:\Program Files\Microsoft Team Foundation Server 14.0\Application Tier\Web Services\_tfs_resources\TestManagement\v1.0\Transforms\1033\TestSuite.xsl`.
2. Modify the `TestSuite.xsl` file in the `<installation path>\Application Tier\Web Services\_tfs_resources\TestManagement\v1.0\Transforms\<locale>\TestSuite.xsl` folder.
3. Open the `TestSuite.xsl` file in Notepad and add the following lines of code to include your company logo into the export template:

```
<div style="align:center;">
<img src="../../_static/tfs/18/_content/companylogo.png" />
</div>
```

The results of the customization can be tested by generating an export through the Test Hub.

Charting testing status on the dashboard in team portal

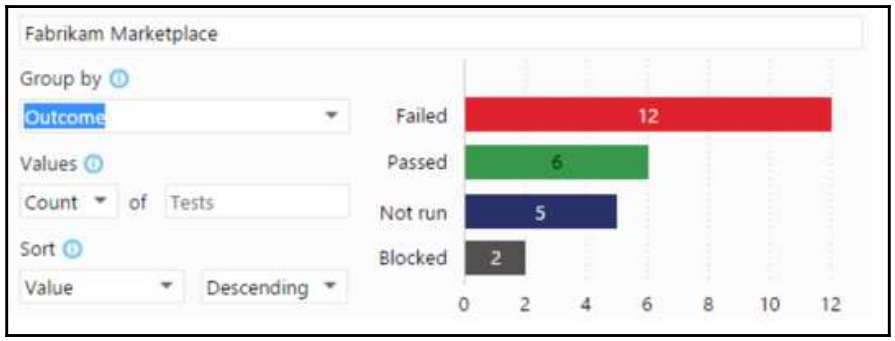
The charting tools in team portal provide a great way to analyze and visualize test case execution. The charts created through the charting tools can be pinned to custom dashboards. Both charts and dashboards are fantastic information radiators to share the test execution results with team members and stakeholders. In this recipe, we'll learn how to pin the test execution results on a custom dashboard in a team portal.

Getting ready

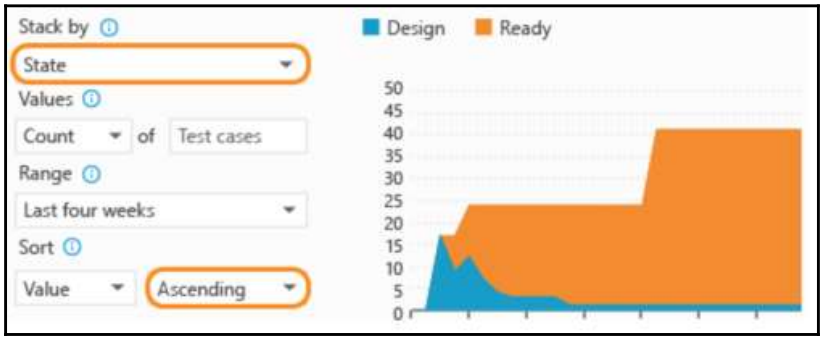
Follow the steps in the *Configuring dashboards in Team Project* recipe in `Chapter 1`, *Planning and Tracking Work*, to create a custom dashboard for testing.

How to do it...

1. Navigate to the Test Hub in the Parts Unlimited team project. The Test plan page gives you a list of test suites and a list of test cases for the selected suite. The **Charts** tab gives you a great way to visualize this information.
2. Click on the **+** icon and select **New test result charts**.
3. Select a bar chart and **Group by** as **Outcome**: this renders the test case outcome in the bar chart. Click **OK** to save the chart.
4. Right-click the newly created chart and pin the chart to the testing dashboard:



5. Now click on the **+** icon and select **New test case chart**. Test case chart types support trend charts; the supported trend period is from seven days to up to 12 months.
6. Select the stacked area chart type and choose to stack by State. This will allow you to visualize the state of the test cases over time.
7. Click **OK** to save the chart. Right-click the chart and pin to the dashboard:



How it works...

The charts are calculated using the Work Item data. When Work Items are updated, the charts reflect the updates immediately. To learn more about the charting functionality in a team web portal, refer to the walk-through here: `http://bit.ly/1PGP8CU`.