3
Build and Release Agents

The build system in Azure DevOps Server known as Azure Pipelines is an open source, cross-platform, extensible, task-based execution system with a rich web interface that allows us to author, queue, and monitor builds. The new JSON-based build system was first introduced in TFS 2015, and it has since been rewritten for the .NET Core CLR as one code base in C#. The modern platform continues to evolve through the open source ecosystem, with new features and enhancements rolling out every other week. The build system is set to evolve further with the new multi-phase builds and a build-definition-as-code functionality that was introduced through YAML-based builds in Azure DevOps. These features have been recently introduced into Azure DevOps Server with the update 1 of Azure DevOps server 2019. In the following screenshot you can see the evolution of the build system over the last 10 years:

Generation	Name	Configuration	Introduced In
Generation 1	MS Build	XML	TFS 2005
Generation 2	XAML Build	WWF	TFS 2010
Generation 3	TFBuild	JSON	TFS 2015

In this chapter, we will cover the following recipes:

- Unattended configuration of build agents using PowerShell
- Downloading agents using the GitHub release API
- Configuring deployment groups
- Configuring the agent to use a proxy
- Analyzing build usage data
- Automating agent pool maintenance
- Configuring build and release retention policies
- Agent capabilities and build demands for special builds
- Managing agent permissions using role-based access

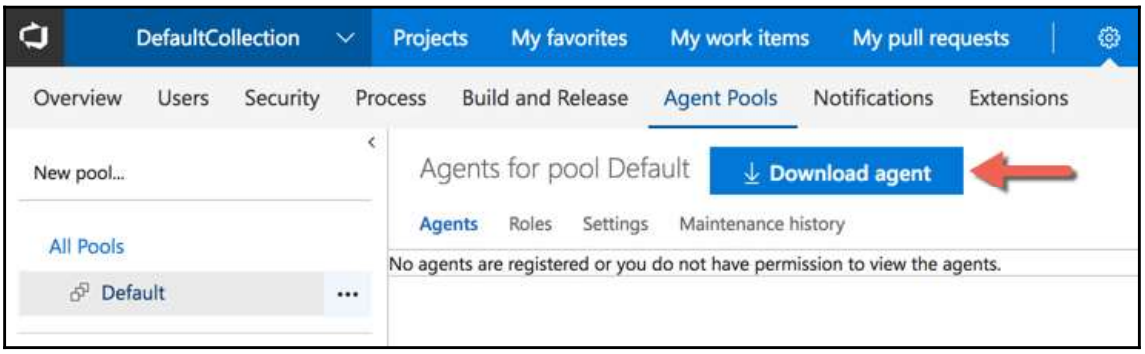
Unattended configuration of build agents using PowerShell

Azure DevOps Server Build and Release agents are the engines of your build system; the size of the infrastructure translates to the speed at which you can run and scale the build process. As you ramp up the use of the build system to automate Continuous Integration pipelines, you are going to need more agents. An automated process to add and remove build agents allows you to scale up and scale down the agents on demand. The build system has native support for unattended installation. In this recipe, we'll learn how to configure a build agent programmatically in an unattended mode using PowerShell.

Getting ready

To configure a build agent, you should be a member of the build administrators group and an administrator on the target machine. If the target machine is Windows 10 or beyond (x64), all the prerequisites will already be in place. If the target machine is Windows 7 to Windows 8.1, or Windows Server 2008 R2 SP1 to Windows Server 2012 R2 (64-bit), you will need to ensure that PowerShell version 3 or newer is available on the target system. Even though not technically required by the agent, many build scenarios require that Visual Studio be installed to get all the tools. It is recommended that you use Visual Studio 2015 or later.

Microsoft has open sourced its build system on GitHub under the `Microsoft/azure-pipelines-agent` project name. You can download the latest version of the agent directly from the GitHub repository (`https://github.com/Microsoft/azure-pipelines-agent`) or from the **Agent Pools** page under the collection administration page in the team portal:



How to do it...

1. Launch PowerShell in elevated mode and execute the following command:

```
> mkdir tfs_a1 && cd tfs_a1

tfs_a1> Add-Type -AssemblyName System.IO.Compression.FileSystem
;
[System.IO.Compression.ZipFile]::ExtractToDirectory("$HOME\Down
loads\vsts-agent-win-x64-2.129.0.zip", "$PWD")
```

In the preceding command, replace the version of the agent (`vsts-agent-win-x64-2.129.0.zip`) with the version you intend to configure.

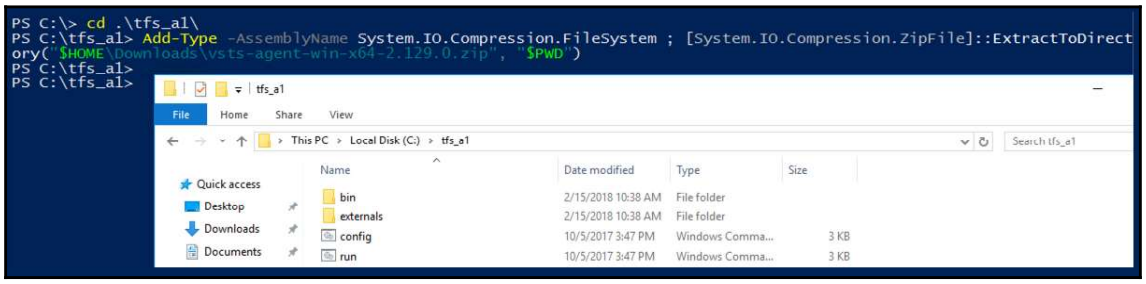
2. Configure the agent to run as a Windows service:

```
tfs_a1> .\config.cmd --unattended `
>> --url http://tfs2018.westeurope.cloudapp.azure.com/tfs `
>> --auth pat --token xxxxxxxx `
>> --pool default --agent tfs_a1 `
>> --runAsService --windowsLogonAccount contoso\zz_tfs-build --
windowsLogonPassword xxxxx

>> Connect:
```

```
Connecting to server ...

>> Register Agent:

Scanning for tool capabilities.
Connecting to the server.
Successfully added the agent
Testing agent connection.
2018-02-15 11:05:52Z: Settings Saved.
```

How it works...

In the preceding command, we are simply creating a new directory and then instructing the ZIP file to be extracted in this new directory:

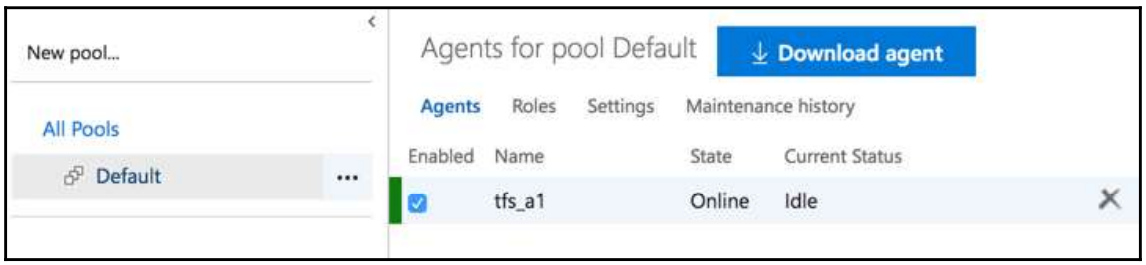


To configure the agent in unattended mode, all the configuration for the installation needs to be specified through the command-line switches. In the `--unattended` command, we are simply passing the details of the Azure DevOps server's URL, the type of authentication to use, and the pool the agent needs to be configured into. When selecting the authentication type as PAT, you'll need to pass the PAT account that will be used by the agent to authenticate with TFS. In addition, you have the option of running the agent as a Windows service under a Windows domain account, which is what we are passing through in the `--runAsService`, `--windowsLogonAccount`, and `--windowsLogonPassword` switches.

In order to use basic authentication while configuring the agent, you need to have a secure connection (SSL) with the TFS server. If you don't have a secure connection, the preceding command will fail with an error message: **Basic authentication requires a secure connection to the server**.

If you do not have SSL configured for your TFS server, you can configure the agent using integrated authentication. Once the command has been successfully executed, you'll see the agent show up in the Agent Pools page, as shown in the following screenshot:

```
tfs_a1> .\config.cmd --unattended `
>> --url http://tfs2018.westeurope.cloudapp.azure.com/tfs `
>> --auth integrated `
>> --pool default `
>> --agent tfs_a1
```



Downloading agents using the GitHub release API

In a big move to embrace open source, Microsoft transitioned a lot of its key projects to GitHub. By developing products in an open source and contributing back to the open source communities, Microsoft is starting to change its negative public perception. This has resulted in some very surprising partnerships and an overall growth story for Microsoft, which is reflected in its stock price going up significantly over the last couple of years.

The `azure-pipelines-agent` and `azure-pipelines-tasks` projects are also hosted on GitHub. *How does this benefit you?* You can see all of the product's code, see the quality and architecture of the patterns used, have visibility of the product roadmap, contribute to the product's development, and engage with the product team by raising feedback and issues through GitHub. Both experimental and long-term supported versions of the agents are released on GitHub. Based on the pace at which the product is evolving, it is likely that the agent version you are running today will be superseded by a newer version tomorrow with more desirable features. Luckily, the GitHub release API supports programmatic invocation, so you never have to manually check for updates.

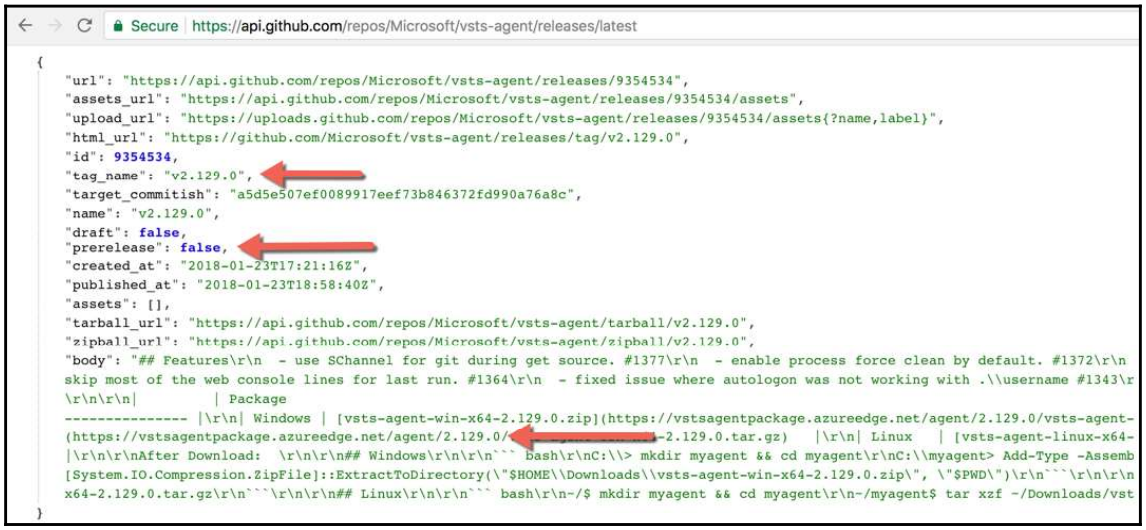
In this recipe, we'll learn how to use PowerShell to query the GitHub release API for the latest version of the agent. Upon finding a long-term supported version of the agent, how to download the agent in a designated folder path. You can optionally extend this solution to include the PowerShell script from the previous recipe to create an end-to-end automated process for downloading, unpacking, and installing agents programmatically using PowerShell scripting.

Getting ready

The release API for GitHub is well-documented at `https://developer.github.com/v3/repos/releases/`. The API supports various functions, including the ability to get all releases for a repository, get a specific release, get a release by tag name, and most importantly, to get the latest release. In this recipe, we'll be using the get latest release functionality:

```
https://api.github.com/repos/Microsoft/azure-pipelines-agent/releases/latest
```

If you invoke this URL in a browser, you'll get a JSON response that includes most of the properties we'll be leveraging in our PowerShell script. The body of the response also includes the `https://vstsagentpackage.azureedge.net/agent` URL. A download URL can be dynamically generated for the platform of your choice using the version of the release derived using `tag_name`:



How to do it...

1. Launch PowerShell, and then use the `Invoke-RestMethod` cmdlet to call the GitHub release API to get the latest release of the agent:

```
# Get the latest release of the agent from the GitHub API
$latestRelease = Invoke-RestMethod
                 -Uri
"https://api.github.com/repos/Microsoft/azure-pipelines-agent/r
eleases/latest"
Value of "tag_name" : "v2.129.0"
```

2. The `tag_name` property shows the name of the tag. By simply removing the first character, you'll get the version number of the agent:

```
$v = $latestRelease.name.Substring(1,
$latestRelease.tag_name.Length-1)
Value of $v : "2.129.0"
```

3. Dynamically construct the URL needed to download the agent. As you can see in the command below the string is being concatenated to create the download URL of the agent for the Windows platform:

```
$latestReleaseDownloadUrl =
"https://vstsagentpackage.azureedge.net/agent/" `
+ $v + "/vsts-agent-win-x64-" + $v + ".zip"

Value of $latestReleaseDownloadUrl =
https://vstsagentpackage.azureedge.net/agent/2.129.0/vsts-agent
-win-x64-2.129.0.zip
```

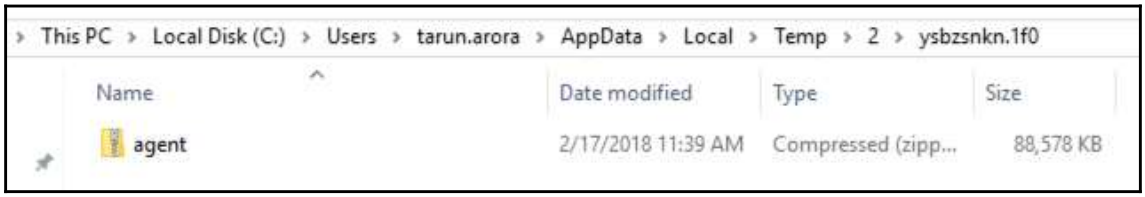
4. Create a new temporary folder; if it doesn't exist, force the creation of the temporary directory:

```
$agentTempFolderName = Join-Path
$env:temp([System.IO.Path]::GetRandomFileName())

If(!(test-path $agentTempFolderName))
{
    New-Item -ItemType Directory -Force -Path
$agentTempFolderName
}
```

5. Call the `Invoke-WebRequest` cmdlet with the agent-download URL to download the agent into the newly created temporary directory:

```
# Download the agent to the temp directory
Invoke-WebRequest -Uri $latestReleaseDownloadUrl -Method Get `
-OutFile "$agentTempFolderName\agent.zip"
```

6. The `agent` will be downloaded into the newly created temporary folder:



How it works...

Bringing it all together, in this section we'll look at the complete script and how it works. The script is wrapped up in a try catch block for error handling, and a retry procedure has been added for resilience. In the following script, the `Invoke-RestMethod` cmdlet is used to get the latest version of the agent from GitHub. The result is then consumed to generate a download installer for the Windows-based agent. The agent is downloaded to a temporary folder and the script checks whether the temporary folder is already in place; if not, it creates the temporary folder before using the `Invoke-WebRequest` cmdlet to download the agent to the temporary folder:

```
$retryCount = 3
$retries = 1
$agentTempFolderName = Join-Path
$env:temp([System.IO.Path]::GetRandomFileName())
Write-Verbose "Downloading Agent install files" -verbose
do
{
    try
    {
        Write-Verbose "Trying to get download URL for latest agent
release..."
        # Get the latest release of the agent from the GitHub API
        $latestRelease = Invoke-RestMethod -Uri `
"https://api.github.com/repos/Microsoft/vsts-agent/releases/latest"
        # Format the name to create a download URL for windows
        $v = $latestRelease.name.Substring(1, $latestRelease.name.Length-1)
        $latestReleaseDownloadUrl =
```

```
"https://vstsagentpackage.azureedge.net/agent/" `
                                        + $v + "/vsts-agent-win-x64-" + $v
+ ".zip"

        # Validate that the temp directory exists or create it
         If(!(test-path $agentTempFolderName)){
              New-Item -ItemType Directory -Force -Path
$agentTempFolderName
        }
        # Download the agent to the temp directory
        Invoke-WebRequest -Uri $latestReleaseDownloadUrl `
                                     -Method Get -OutFile
"$agentTempFolderName\agent.zip"
        Write-Verbose "Download agent successfully on attempt $retries" -
Verbose
        break
    }
    catch
    {
        $exceptionText = ($_ | Out-String).Trim()
        Write-Verbose "Exception occurred downloading agent: `
                                   $exceptionText in try number $retried" -
Verbose
        $retries++
        Start-Sleep -Seconds 30
    }
}
while ($retries -le $retryCount)
```

In case there is an exception, the exception is caught and there is logic to retry the download. A total of 3 retries are attempted at an interval of 30 seconds before exiting the script.

Configuring deployment groups

An application environment is composed of multiple servers in different roles, such as web, application, and database. Scaled out versions of these environments could have multiple servers front-ended by load balancers and availability groups. While agents in agent pools give you a way to deploy your application, you are responsible for bringing together the agents in the agent pool to deploy to your environment.

In this model, you are responsible for managing the complexity of how the deployment impacts the environment, such as orchestrating the rotation of the web servers as they are being upgraded. In this model, it's hard to answer simple questions such as the version of the release deployed on a machine. Microsoft has significantly enhanced the machine groups feature that it first introduced in TFS 2015 and rebranded as deployment groups.

Simply put, deployment groups are a collection of agents collectively representing an application environment, such as Dev, UAT, Pre-Prod, or Production. Each machine in the deployment group has an agent; metadata can be associated with the agent by adding tags. The deployment group can then be queried for this tag to return a list of agents that match the tags. This makes deploying to a multi-server web tier very easy. As the framework is aware of all the agents with the `WebServer` tag, you can specify a deployment rule to roll out the deployment on a small subset of web servers and stop in case of any failures. All the native agent capabilities are still available to you – for example, you can view live logs for each server as a deployment takes place, and download logs for all servers to track your deployments to individual machines. The deployment group records the version of the release that was deployed in an environment and on the individual servers in the deployment group as well. Deployment groups also provide a security context, so you can add users and give them appropriate permissions to administer, manage, view, and use the group.

The host machine can have one or more agents deployed on it, and each agent can be associated with a different deployment group. This gives you the ability to use shared environments exclusively through their own deployment groups.

Deployment groups are not visible to build pipelines; they are only meant to be used in release pipelines. While we'll be covering how to use deployment groups in `Chapter 7`, *Azure Artifacts and Dependency Management*, we'll learn how to configure the Azure DevOps agent into a deployment group.

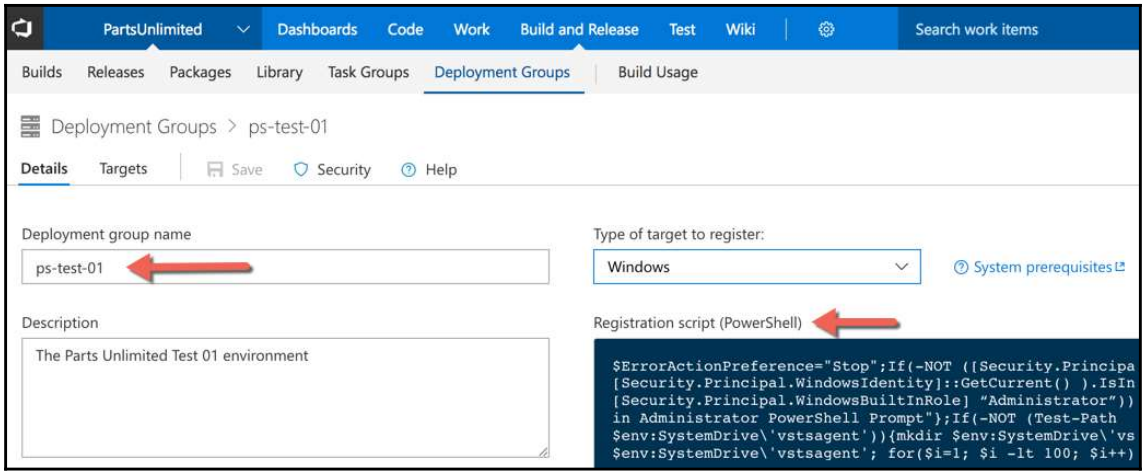
Getting ready

To create a deployment group, you need to be a member of the build administrator and release administrator group; membership to the project-collection administrator group also gives you permission to perform this action across multiple team projects in a collection.

How to do it...

In this section we'll cover the steps to setup an agent into a deployment group:

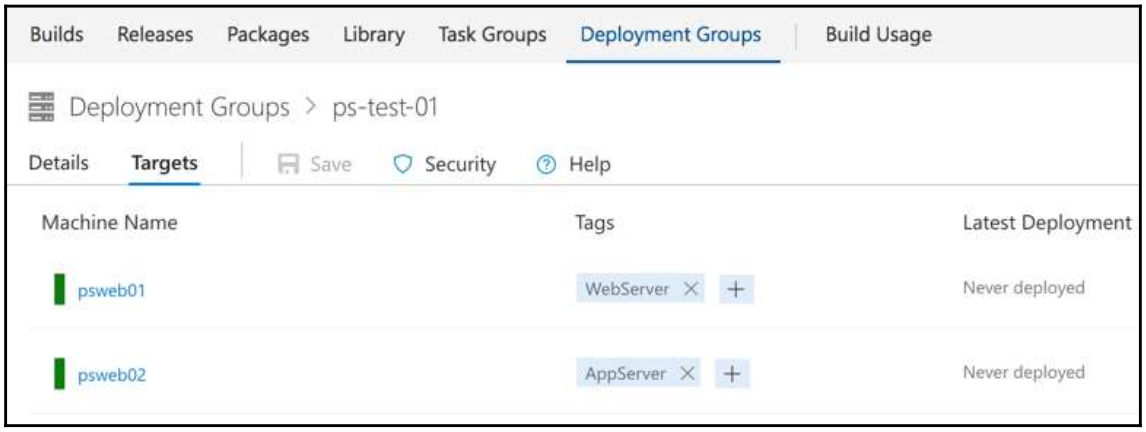
1. Navigate to the build and release hub in the PartsUnlimited team project portal. Click on **Deployment Groups** and add a new deployment group, `ps-test-01`:



2. Copy the PowerShell script, then navigate to the target machine you intend to add to this deployment group. Run PowerShell in elevated mode and execute the script. The script downloads the agent and configures the agent as a deployment group. Run this script on other machines you intend to join this deployment group:

```
PS C:\vstsagent\A4> $ErrorActionPreference="Stop";If(-NOT ([Security.Principal.WindowsPrincipal][Security.Principal.Windo
wsIdentity]::GetCurrent() ).IsInRole( [Security.Principal.WindowsBuiltInRole] "Administrator")){ throw "Run command in Ad
ministrator PowerShell Prompt"};If(-NOT (Test-Path $env:SystemDrive\'vstsagent')){mkdir $env:SystemDrive\'vstsagent'}; cd
 $env:SystemDrive\'vstsagent'; for($i=1; $i -lt 100; $i++){$destFolder="A"+$i.ToString();if(-NOT (Test-Path ($destFolder)
)){mkdir $destFolder;cd $destFolder;break;}}; $agentZip="$PWD\agent.zip";(New-Object Net.WebClient).DownloadFile( 'https:
//go.microsoft.com/fwlink/?linkid=858950', $agentZip);Add-Type -AssemblyName System.IO.Compression.FileSystem;[System.IO.
Compression.ZipFile]::ExtractToDirectory( $agentZip, "$PWD");.\config.cmd --deploymentgroup --agent psweb02 --runasservic
e --work '_work' --url 'http://localhost/tfs/' --collectionname 'DefaultCollection' --projectname 'PartsUnlimited' --depl
oymentgroupname "ps-test-01" --auth Integrated; Remove-Item $agentZip;


    Directory: C:\vstsagent


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        2/18/2018   4:45 PM                A5
```

3. Flip over to the **Targets** tab for the `ps-test-01` deployment group in the PartsUnlimited team portal. This will show you the list of all the target machines joined into this deployment group:



How it works...

Let's look at the deployment-group-registration script in more detail. The error preference for this script is set to stop, implying the script will stop execution on the first failure:

```
$ErrorActionPreference="Stop";
```

Validate that the user executing the script is part of the administrator role on the host machine:

```
If(-NOT
([Security.Principal.WindowsPrincipal][Security.Principal.WindowsIdentity]:
:GetCurrent() ).IsInRole( [Security.Principal.WindowsBuiltInRole]
“Administrator”))
{ throw "Run command in Administrator PowerShell Prompt"};
```

Validate that the `vstsagent` folder exists in the `System` directory on the host machine. If not, create a new directory for `vstsagent` and navigate to that directory. If an agent already exists in the directory, create a new `agent` folder with a different name:

```
If(-NOT (Test-Path $env:SystemDrive\'vstsagent'))
{mkdir $env:SystemDrive\'vstsagent'};
cd $env:SystemDrive\'vstsagent';
```

```
for($i=1; $i -lt 100; $i++){$destFolder="A"+$i.ToString();
if(-NOT (Test-Path ($destFolder))){mkdir $destFolder;cd
$destFolder;break;}};
```

Download `agent.zip` and unzip it into the newly created `agent` folder:

```
$agentZip="$PWD\agent.zip";
(New-Object Net.WebClient).DownloadFile(
'https://go.microsoft.com/fwlink/?linkid=858950', $agentZip);
Add-Type -AssemblyName
System.IO.Compression.FileSystem;[System.IO.Compression.ZipFile]::ExtractTo
Directory( $agentZip, "$PWD");
```

Configure the agent as a deployment group, use the host machine name to name the agent, and configure the agent to run as a Windows service. Join the agent in the `ps-test-01` deployment group in the `PartsUnlimited` team project:

```
.\config.cmd --deploymentgroup --agent $env:COMPUTERNAME --runasservice --
work '_work' --url 'http://tfs2018.westeurope.cloudapp.azure.com/tfs/' --
collectionname 'DefaultCollection' --projectname 'PartsUnlimited' --
deploymentgroupname "ps-test-01" --auth Integrated;
Remove-Item $agentZip;
```

Remove the agent installer file from the `agent` directory:

```
Remove-Item $agentZip;
```

You can use the `--unattended` switch we saw in the *Unattended configuration of build agents using PowerShell* recipe to configure the deployment group in unattended mode. This is useful if you want to push the configuration of deployment groups remotely on target machines.

Configuring the agent to use a proxy

Enterprises that host their infrastructure on-premise or in a hybrid cloud setup tend to use a multi-level security approach. This usually involves one or more firewalls that protect the infrastructure from external traffic, and a web proxy to control the intranet and internet traffic. In such a scaled setup, traffic generated from the agent may be blocked from connecting to the Azure DevOps server if it is not routed through the proxy. Luckily, the Azure DevOps agent infrastructure supports proxy configuration natively. In this recipe, we'll learn how to configure a web proxy during agent configuration.

Getting ready

Ensure you've downloaded the latest version of the agent locally on the target machine where you intend to install and configure the agent. You'll need to be part of the build administrators group in Azure DevOps Server to be able to connect the agent as well as an administrator on the target machine to be able to install the agent. The TFS agent is programmed to pick up the proxy settings configured in the `.proxy` file in the `agent` folder. Therefore, for the agent to pick up the proxy settings, you will need to create a `.proxy` file in the agent install folder ahead of configuring the agent.

How to do it...

1. Navigate to the location where the agent has been downloaded and unzipped:

```
> cd c:\tfs_a1
tfs_a1>
```

2. Create a `.proxy` file with the proxy URL:

```
echo http://theProxyServer:443 > .proxy
```

The preceding configuration is sufficient if the proxy doesn't require authentication. This is usually the case if you are using a Windows domain account to run the agent. However, if you are running the agent under the default network credential or the user running the agent is required to authenticate with a proxy, you'll need to perform the following additional steps.

3. Provide the proxy credentials to TFS using environment variables. TFS treats these credentials as secrets and then masks the values before passing them into the job output:

```
set VSTS_HTTP_PROXY_USERNAME=myProxyUser
set VSTS_HTTP_PROXY_PASSWORD=myProxyPassword
```

How it works...

With the proxy configured in the `root` folder of the build agent, you can use `config.cmd`, as demonstrated in the recipe Unattended configuration of build agents using PowerShell, to configure the agent. The agent automatically picks up the proxy configuration; if the proxy configuration requires authentication, the agent uses the `VSTS_HTTP_PROXY_USERNAME` and `VSTS_HTTP_PROXY_PASSWORD` environment variables to authenticate with the proxy. Any build jobs that you run thereafter will print the proxy URL being used by the agent in the build output.

Analyzing build usage data

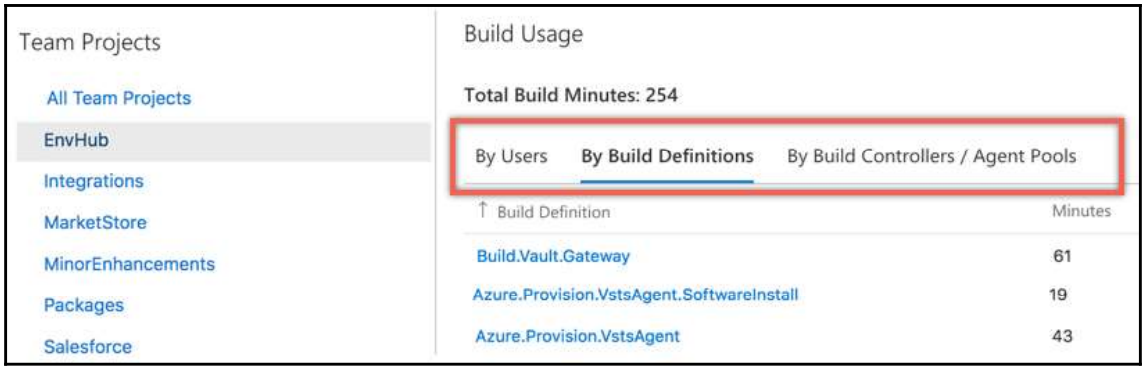
Wouldn't it be great if you could get insights into how the build system is being used? For example, which projects are using builds more than others, which definitions are consuming most build time, and top build users. After all, data is the new currency, and you should feel empowered to make empirical data-driven decisions. This is especially useful if you plan to evolve the build pools and queues with empirical usage data trends. In this recipe, we'll learn how to get insights into the usage of the build system using an open source extension available in the marketplace.

Getting ready

The marketplace features the **Build Usage** extension (`https://marketplace.visualstudio.com/items?itemName=ms-devlabs.BuildandTestUsage`). This open source extension, released by the Microsoft DevLabs team, provides insights into the build infrastructure usage at different granularities. Install this extension in the project collection you plan to analyze the data for.

How to do it...

1. Navigate to the collection administration page and open the **Build Usage** page. **All Team Projects** gives you a headline of the total build usage across the collection. You can drill into the specific team projects to see the build usage by users, definitions, and agent pools. The data can be filtered to this month, last month, the last six months, or a custom period:



2. Click the **Export** button to export the results into a CSV file, which you can then use for further analysis offline. In addition, the extension also includes a dashboard widget.

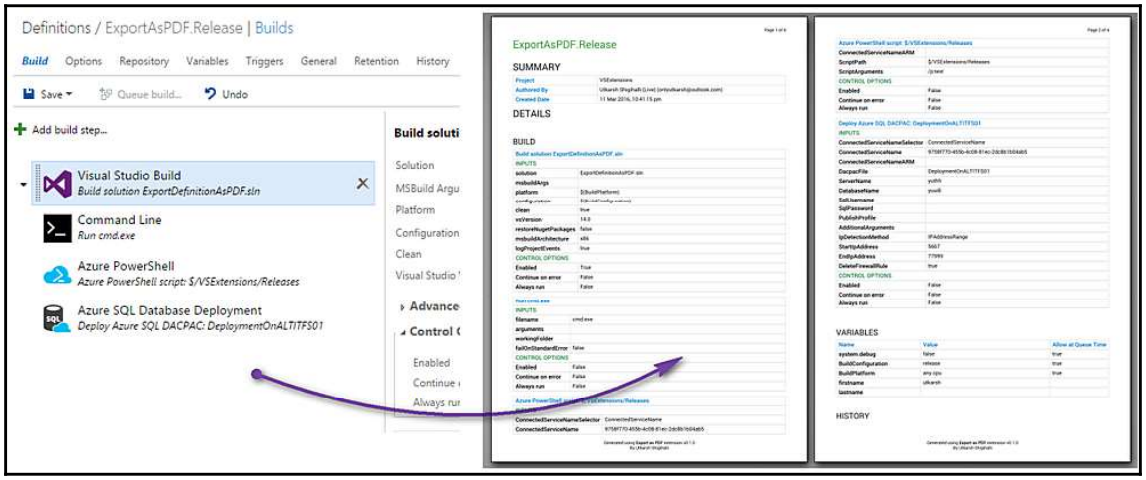
How it works...

The extension is open source; you can take a look at the extension's code implementation on GitHub at: `https://github.com/ALM-Rangers/build-usage-widget-extension/blob/master/vsts-buildusage/src/build.ts`. The extension uses the build REST API to get a list of all the build definitions across the collection and then all builds recorded against those definitions. It then processes this data by aggregating the properties to show a view of usage by projects, definition, user, and agent pool:

```
public getBuildExecutions(projectId: string, startDate: Date, finishDate: Date, successCallback, errorCallback): void {
    let builds = this.getBuilds(projectId, startDate, finishDate);

    builds.then(
        (builds) => {
            let totalTime: number = 0;
            let buildDataSource = new Array<BuildDetails>();

            for (let i = 0; i < builds.length; i++) {
                let build: Tfs_Build_Contracts.Build = builds[i];
                let buildDefinition: string = build.definition.name;
                let buildController: string = (build.controller == null ? build.queue.pool.name : build.controller.name);
                let minutes: number = (build.finishTime.valueOf() - build.startTime.valueOf()) / 1000 / 60;
                totalTime += minutes;

                buildDataSource.push(new BuildDetails(build.requestedFor.displayName, minutes, buildDefinition, buildContro
            }
            successCallback({ totalTime: totalTime, buildDataSource: buildDataSource });
        },
        (error) => {
            errorCallback(error);
        }
    );
}
```

See also

The Azure DevOps marketplace also features the *Export as PDF* extension (`https://marketplace.visualstudio.com/items?itemName=onlyutkarsh.ExportAsPDF`). This free extension, created by Utkarsh Shigihalli, allows you to export the build definition steps, triggers, history, and so on in a neat report so that you can print or share it with colleagues. The extension is especially useful if you intend to document the build setup for training or knowledge-transfer purposes:



Automating agent pool maintenance

Each build and release pipeline creates a directory under the agent working directory to store the source code, artifacts, and test results. Some builds consume more space than others; as projects evolve, some builds are used more than others. As you may have guessed, this results in agent maintenance activity to clear out the agent work directory. While you wouldn't want to remove everything from the directory, you would certainly be looking to remove some of the less-used build folders. Luckily, the agent comes with out-of-the-box support for pool maintenance. In this recipe, we'll learn how to configure the agent-pool maintenance schedule to automatically free up storage from the agent work directory by removing unused build folders.

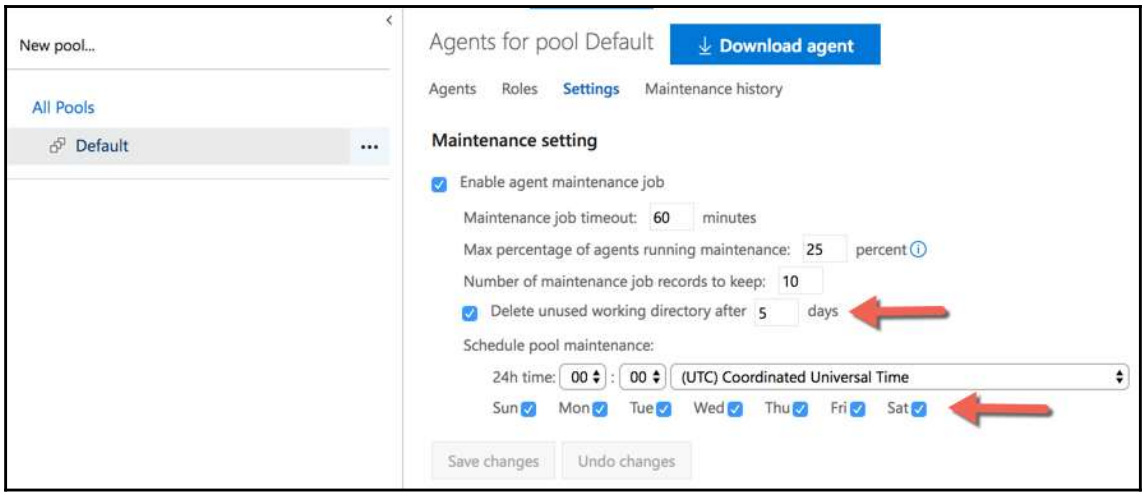
Getting ready

To configure the agent pool maintenance, you need to be part of the build collection administrator group or in the administrator role for the specific agent pool.

How to do it...

In this section we'll look at the steps needed to configure the automated maintenance of the agent pool:

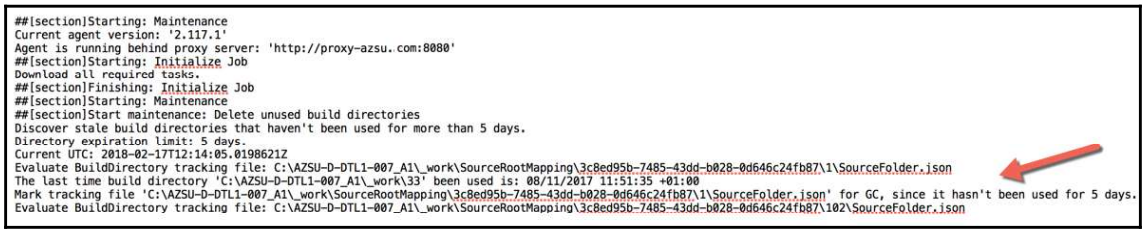
1. Navigate to the collection administration page and open the agent pools page.
2. Select the default agent pool or any other agent pool and click settings to configure the maintenance schedule. Set the schedule to run every day and remove all build and release agent directories that haven't been used in the last **5** days:



How it works…

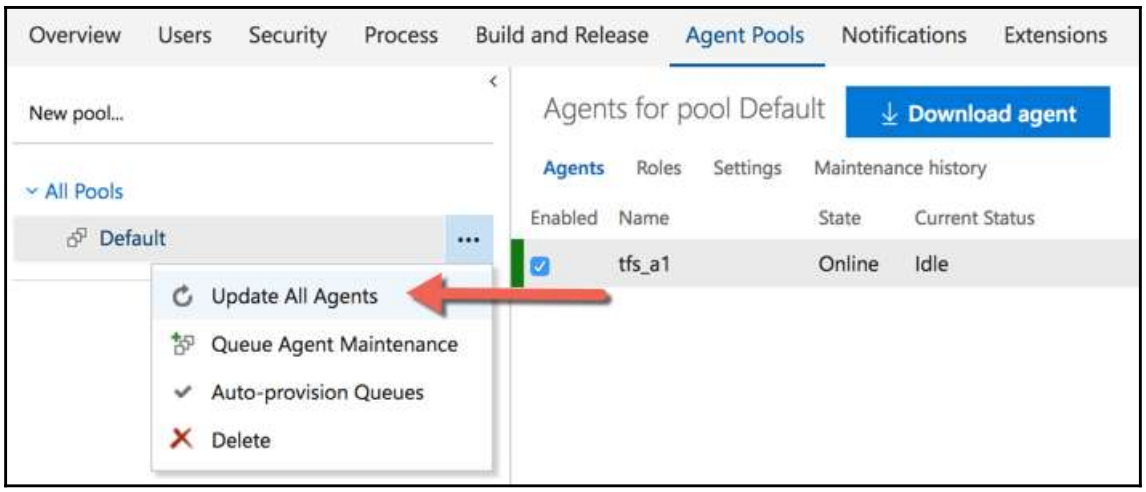
To trigger the maintenance schedule in an ad hoc manner, right click the agent pool, from the context menu select queue agent maintenance. When you trigger agent pool maintenance, this queues a new build job, the results of which are published in the build **Maintenance history**. As per the settings configured, the job only takes the specified percentage of agents out for maintenance, so the pool can still be used for builds and releases:

Since `azure-pipeline-agent` is open source on GitHub, you can see the code of the product. The `RunMaintenanceOperation` function in the `BuildDirectoryManager` class (`https://github.com/Microsoft/azure-pipeline-agent/blob/f9e5bb7337fb51ace995cafeeaa8665cad638a84/src/Agent.Worker/Build/BuildDirectoryManager.cs`) goes through each of the build and release directories under the agent work directory to identify when it was last used. If it meets the criteria, it's marked for garbage collection. All folders marked for garbage collection are deleted, and the overall storage space is reported at the end of the job's execution. If you download the agent maintenance logs, you will see this reflected in the maintenance log:



There's more...

To help you keep the agents updated with the latest product version, Azure DevOps Server offers you the ability to upgrade within the **Agent Pools** page directly, without having to go through the process of removing and reconfiguring agents. To do this, you need to be part of the collection build administrator group or in the administrator role for the agent pool you intend to perform this operation on. From the pool context menu, simply select **Update All Agents**. This will temporarily take the agents in the pool out, download the latest version of the agent, and upgrade in-situ:



Configuring build and release retention policies

In the *Automating agent pool maintenance* recipe, we learned how to configure maintenance schedules on the agent machines. While that helps free up space on the agent machine, there is maintenance activity required on the Azure DevOps Server to free up space by removing unwanted builds and releases. An average build artifact, test results, and associated metadata is in the range of 50 MB.

If the build is run 20 times a day for 30 days, this will generate about 29 GB worth of assets! While Azure DevOps Server does a great job in compressing and storing this data in blob storage, it is best to offload what you don't need. In this recipe, we'll learn how to configure a retention policy for both builds and releases at the collection level to automatically remove builds and releases that match this criteria. We'll also learn how to overwrite the default retention policy for a specific build or release definition.

Getting ready

To administer build resources for the collection, you need to be a member of the Project Collection Build Administrators group.

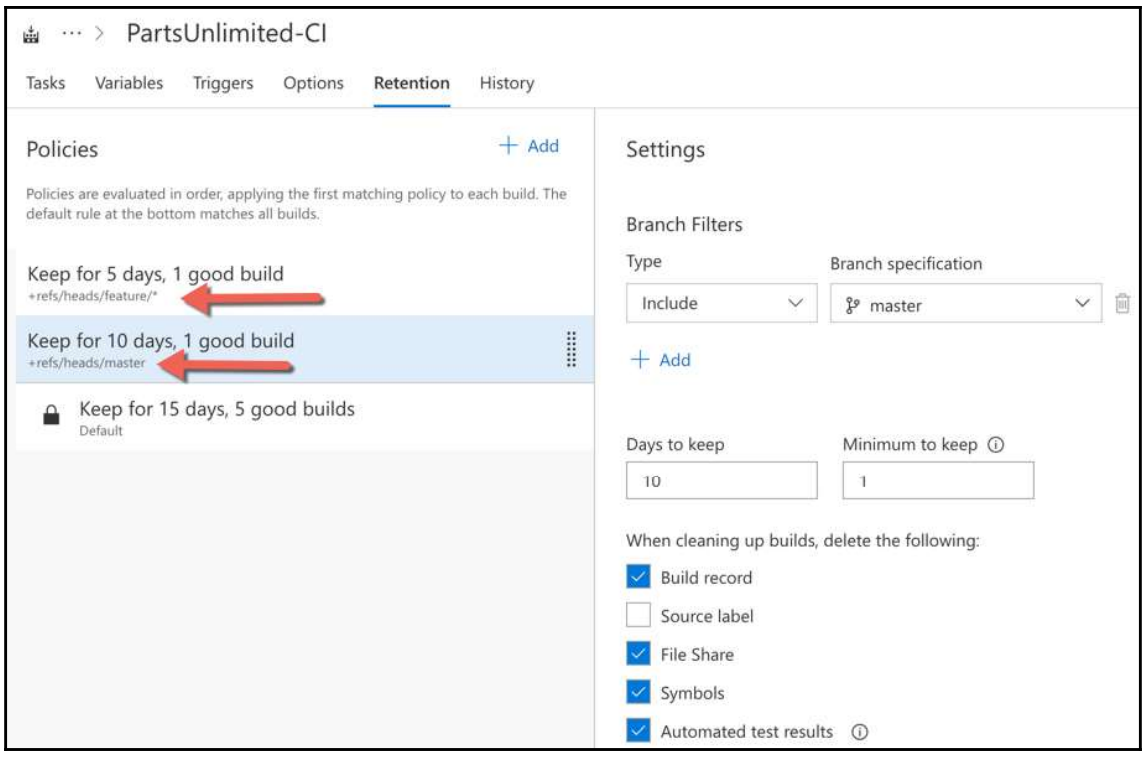
How to do it...

1. Navigate to the collection administration page and open the **Build and Release** page
2. Keep the **Minimum Retention Policy** as is, and change the default configuration for the maximum retention to `15` days and the minimum to keep to `5`
3. Change the days to keep the build record after deletion to `10`

How it works...

TFS has a set of background jobs that are scheduled to run to manage various operations within TFS. You can access the TFS job monitoring page by browsing to `_oi/jobmonitoring page http://<tfsInstance>/tfs/_oi/_jobmonitoring`. The build and retention policy is also orchestrated as a job. Builds and releases that match these criteria, with the exception of those marked for indefinite retention, will be removed.

It is also possible to overwrite the global build and release policy at a build and release level, which can be done from the retention tab in a build or release definition. With the ability to apply branch filters, you can specify different retention schedules for different types of branches. For example, the topic branch builds can be removed more frequently compared to the master branch. You may want to remove the source label for topic branches, but not necessarily for the master branch:



Agent capabilities and build demands for special builds

The build and test execution of an application depends on the specific version of the framework. For example, an application may have a component that depends on DotNetCore 1.0 and another component may depend on DotNetCore 2.0. The build system gives you the ability to define demands in a build definition and specify capabilities in the agent queues. This creates a build that you can route to an agent queue by simply mapping the demand to the capabilities. The framework leverages this capability internally; during the agent setup, the agent collects a list of software and frameworks installed on the host machine. These can be seen in the agent queue or **Agent pools** page under the **Capabilities** tab. In this recipe, we'll learn how to add custom capabilities in the agent queues and demand that in-build definitions target specific agent queues.

Getting ready

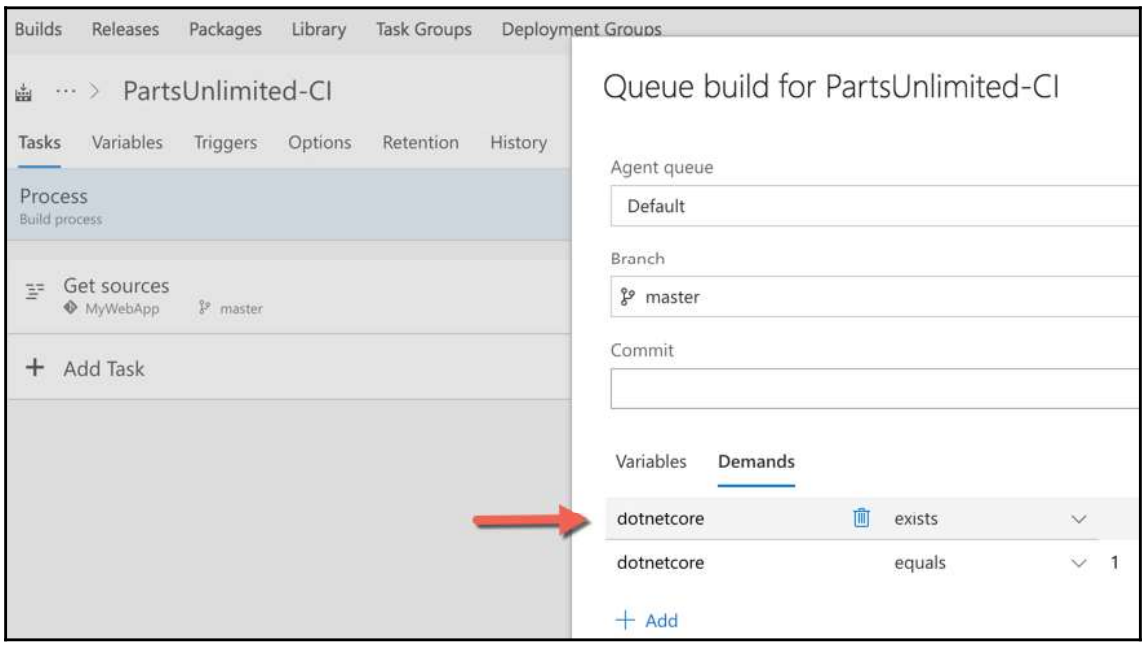
To configure the agent capabilities, you need to be part of the build administrator group or in the administrator role for the specific agent queue.

How to do it...

1. Navigate to the admin portal for the PartsUnlimited team project, select the **Agent queues** page, and click on the **Capabilities** tab. The page displays a list of system capabilities, which shows you a list of all the software and frameworks set up on the host machine. In **USER CAPABILITIES**, click to add a custom capability. This gives you a key-pair; specify the `dotnetcore` name in the key and the number `1` in the value, and then save the changes:

How it works...

When you queue a new build, the build pipeline queries the system capabilities of the queue and determines whether there is an available agent in the queue that meets the build demands. If no agent is found, you will be notified via a warning message. If you queue the build regardless, it will fail at the configured timeout interval if no agent is found that meets the demands in the build definition. At build-queue-time, you have an option to add, remove, or overwrite the build demands:



Managing agent permissions using role-based access

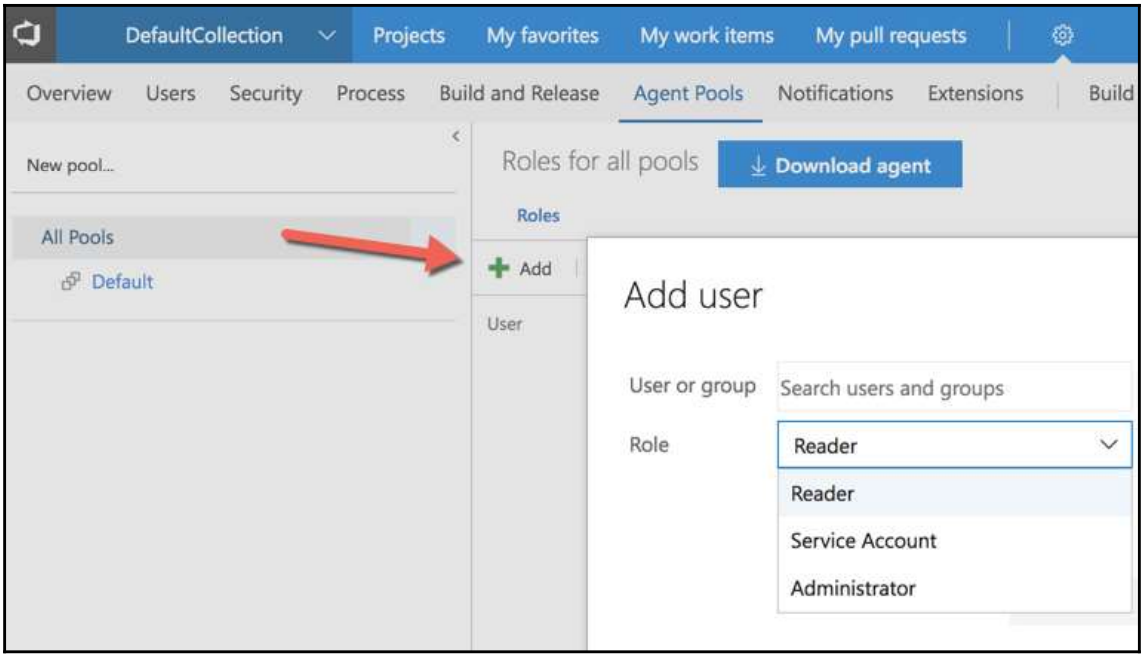
The build system provides *role-based access control* instead of exposing the underlying permissions directly. In this recipe, we'll learn how to permission build resources at the pool and the queue level.

Getting ready

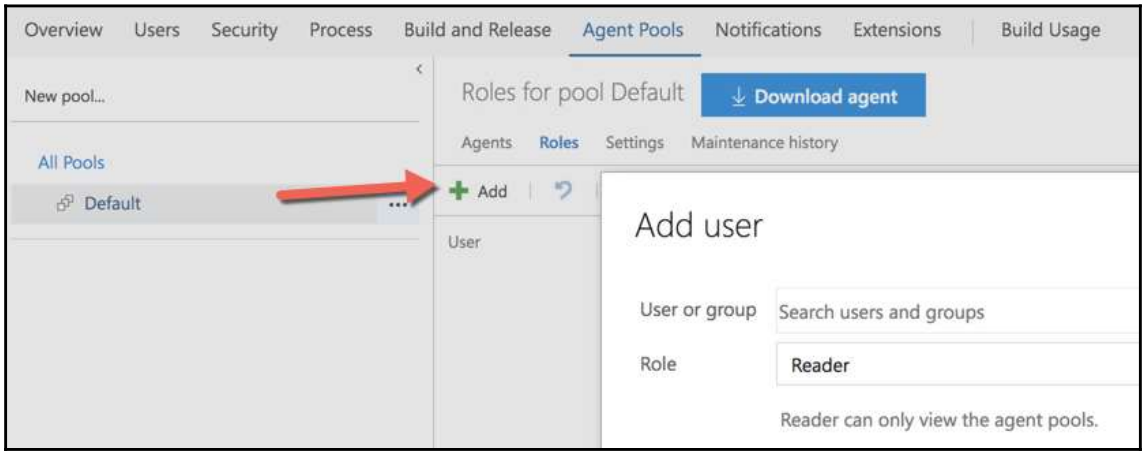
To manage the all pools membership, you need to be a member of the Team Foundation Administrators Group. Membership to the Team Project Collection Administrator Group is required to manage the permissions for individual pools. In order to manage the permissions for the queues, you need to be a member of the Project Collection Build Administrators Group. Build Definition Administration requires membership to the Build Administrators Group.

How to do it...

1. Launch the collection administrator page and navigate to the **Agent pools** page. Click on all pools, then add the service account(s) that you intend to use in the agent pool **Service Account** role:



2. To add a user as a reader to a specific pool, click on the specific pool and add the user account or group to that specific pool:



How it works...

As illustrated in the following diagram, the new build system contains a hierarchical role-based access-control model: