6
Continuous Deployments

Continuous Deployment is the practice of teams to continuously deploy tested and working software to production. The release pipeline of Azure DevOps Server is just an orchestrator of the activities you do on an environment to get your software deployed and running. Another key technique of continuous deployments is the consistency of deployment steps - meaning you follow the same deployment steps across all your deployment environments. The advantage is repeatability, reliability - thus improving your overall delivery so that you release software to production sooner and consistently.

In this chapter, we will see different ways to deploy various types of resources using continuous deployment strategy. Not only will we see how to deploy applications, but will also see how to provision infrastructure so that we eventually achieve repeatable and reliable deployments of our software.

We will cover the following recipes:

- Deploying the database to Azure SQL using the release pipeline
- Consuming secrets from Azure Key Vault in your release pipeline
- Deploying the .NET Core web application to Azure App Service
- Deploying an Azure function to Azure
- Publishing secrets to Azure Key Vault
- Deploying a static website on Azure Storage
- Deploying a VM to Azure DevTest Labs

Deploying the database to Azure SQL using the release pipeline

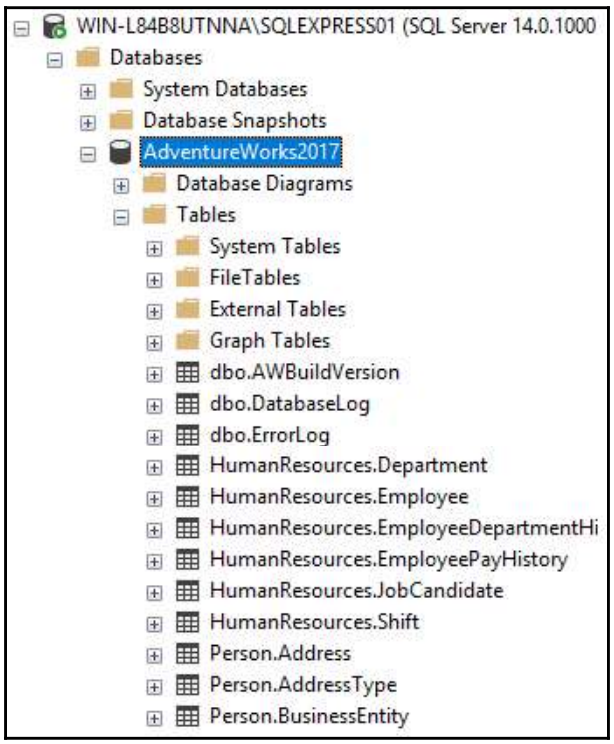
Databases are an integral part of any application and should be part of your DevOps process, which means integrating changes continuously using source control and delivering every change to the environment.

However, most organizations still have a legacy way of deploying databases. Developers still have code stored procedures and commit to the source control, but when it comes to the deployment, a detailed release notes document is prepared on how the database has to be provisioned and handed over to the DBAs.

In this recipe, we will see how we can build a process to consistently develop and deploy the database to Azure SQLDB.

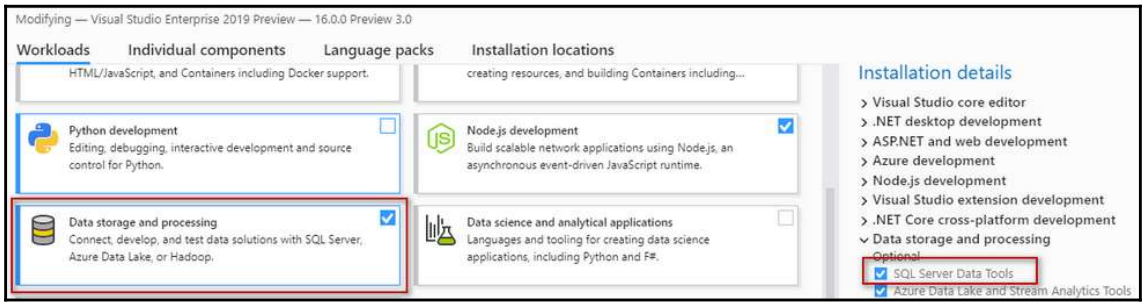
Getting ready

For this recipe, I am using a sample database called **AdventureWorks**, published by Microsoft. If you do not have this database already, Microsoft makes the backup file available for download on GitHub here: `http://bit.ly/2GNpvSo`. Go ahead and download the database as per your SQL Server version. Since I have SQL Server 2017 Express on my machine, I downloaded the `AdeventureWorks2017.bak` file and then restored the database from the backup. Microsoft has instructions on restoring the database, which is documented here: `http://bit.ly/2GKK8hT`. Once you restore you should see the database in SQL Server Management Studio shown as follows:



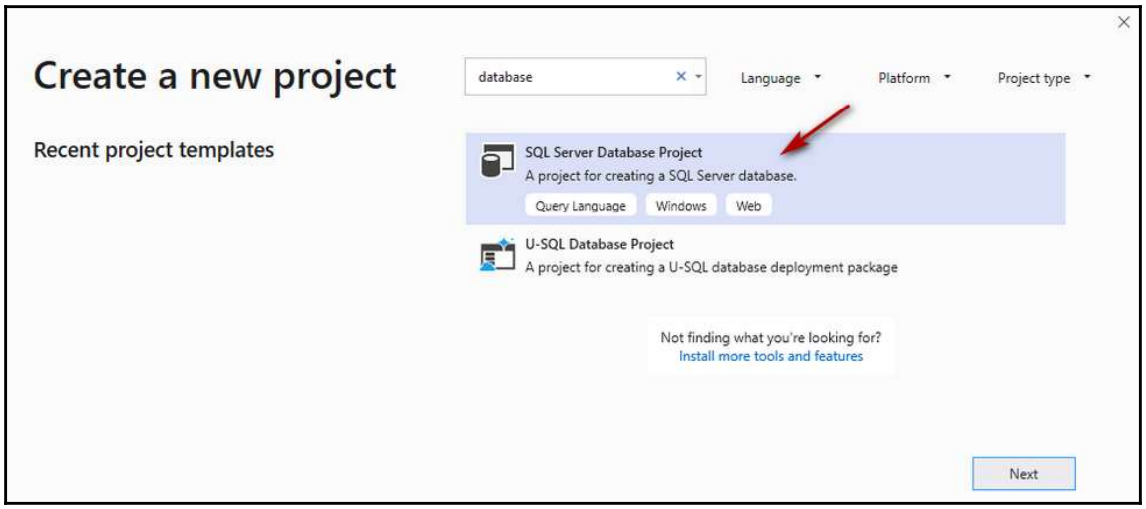
Creating a database project and importing the database

First, we need to ensure that we have **SQL Server Data Tools** (**SSDT**) installed with our Visual Studio version. This tool is available during Visual Studio installation with the data storage and processing workload. It is also available as a standalone installer for Visual Studio:

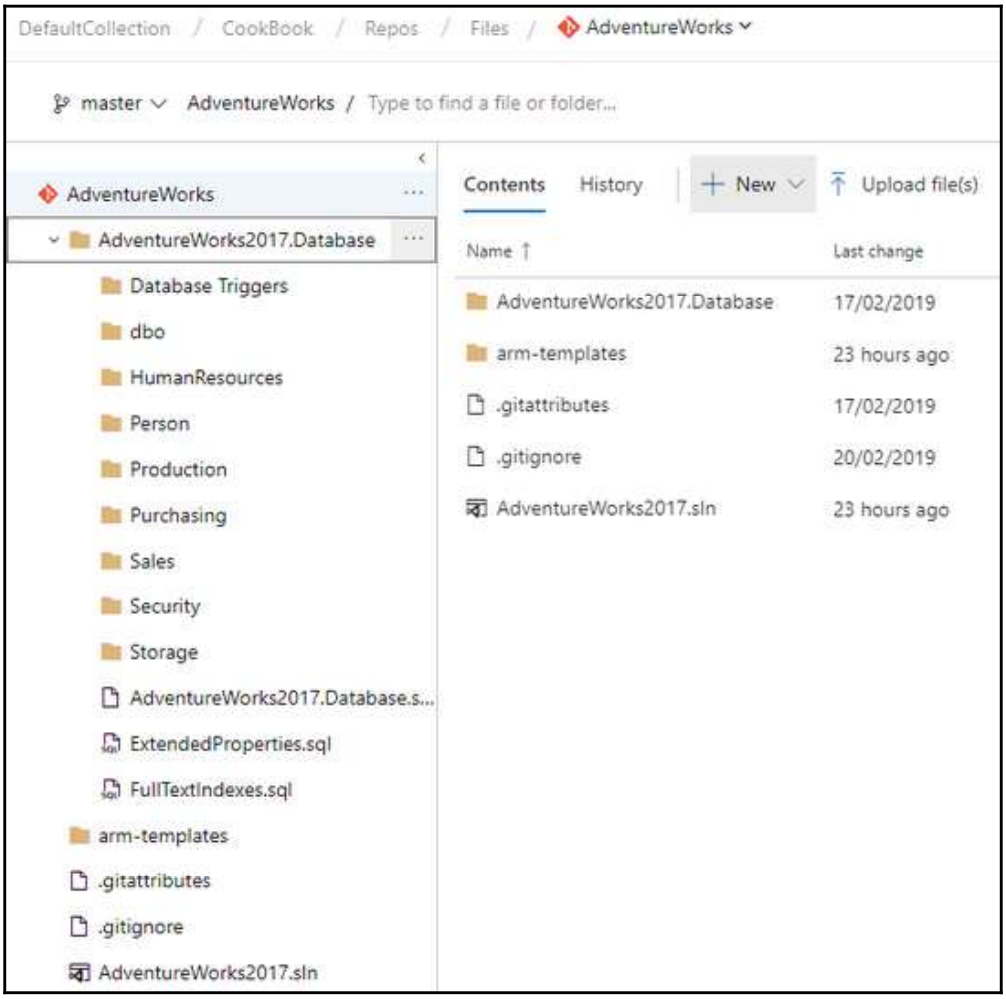


This development tool helps with database design, schema refactoring, and development of database using Visual Studio. Developers can benefit from familiar Visual Studio tools for database development with tools and assistance for code navigation, IntelliSense, debugging, and a rich editor. More information about SSDT can be found in the Microsoft documentation: `http://bit.ly/2tr5Bon`.

Create a new database project, import our existing AdventureWorks database, and commit it to the repository (for more information on how to do this, refer to `http://bit.ly/2tBia0j`):



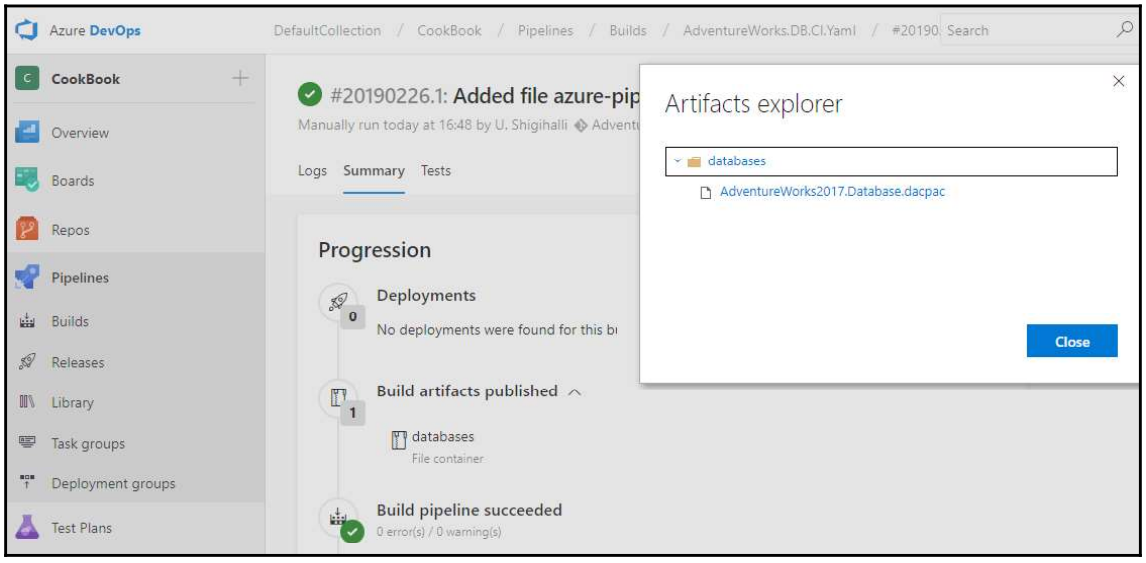
Right click on the project and then import the database. I have then committed the code (project) in my repository, as shown in the following screenshot:



Creating a build definition

Next, we will create a quick build pipeline and produce a `dacpac` package for this database. A database package is a deployable package from your version-controlled database project. You can read more about it at `http://bit.ly/2tr5Bon`.

The build will run and produce the `.dacpac` file, which we will use as the artifact of this build. Now, you have a full continuous integration pipeline so that every time there are any commits in the database project, the build is triggered and the database project is compiled to check your SQL scripts for errors and also verify the schema changes:



How to do it...

Now that we have our build pipeline producing the required artifact, it is time for us to start working on the deployment of this database in Azure. the time of writing, to deploy our existing database, we need to ensure that we have SQL Server provisioned. We could manually create the required SQL Server, but this means we have a manual activity during deployment. The correct solution would be to automate the provisioning of SQL Server as part of the pipeline. This means that even if the required resources are not present, our pipeline will ensure the integrity of the system and create the missing resources (in this case, SQL Server) and then deploy our database. We will be using **Azure Resource Manager** (**ARM**) templates to provision SQL Server in Azure. In simple terms, Azure Resource Manager templates can be used to consistently create/update resources in Azure.

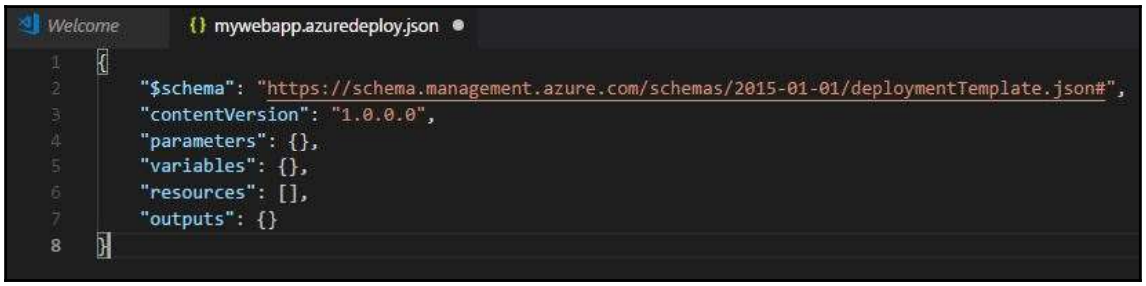
For more information on Azure Resource Manager, visit `http://bit.ly/2FiNtn2`.

As part of the release pipeline, we would like to automate the following:

- The creation of Azure SQL server, if it doesn't exist
- The deployment of the database using the `dacpac` we produced

Creating Azure Resource Manager (ARM) templates

Azure Resource manager templates are simple JSON files that can be used to deploy one or many resources at once. A bare-minimum ARM template is made up of the following structure:



We will create an ARM template to provision the SQL Server on Azure:

1. Using your favorite editor, create a file named `sql.deploy.json`

The ARM templates used here are available under `RCP01-Database-CD` folder.

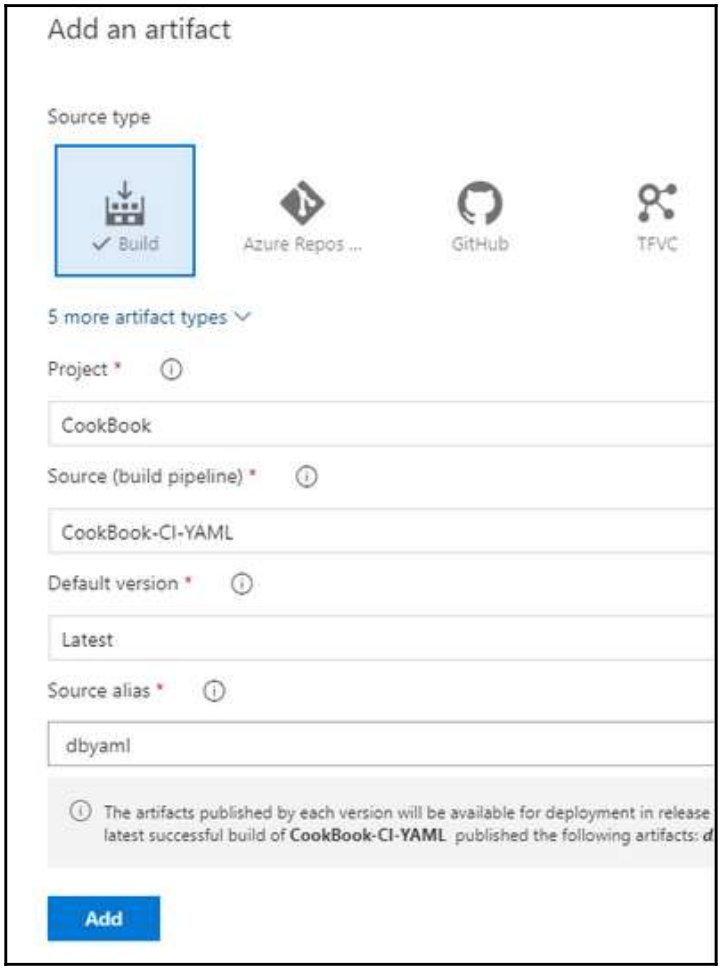
Let's quickly go through what we are doing in the `sql.deploy.json` ARM template. First, in the `parameters` section, we declared a few parameters:

```
{
    "$schema":
"https://schema.management.azure.com/schemas/2015-01-01/deploymentTempl
ate.json#",
    "contentVersion": "1.0.0.0",
```

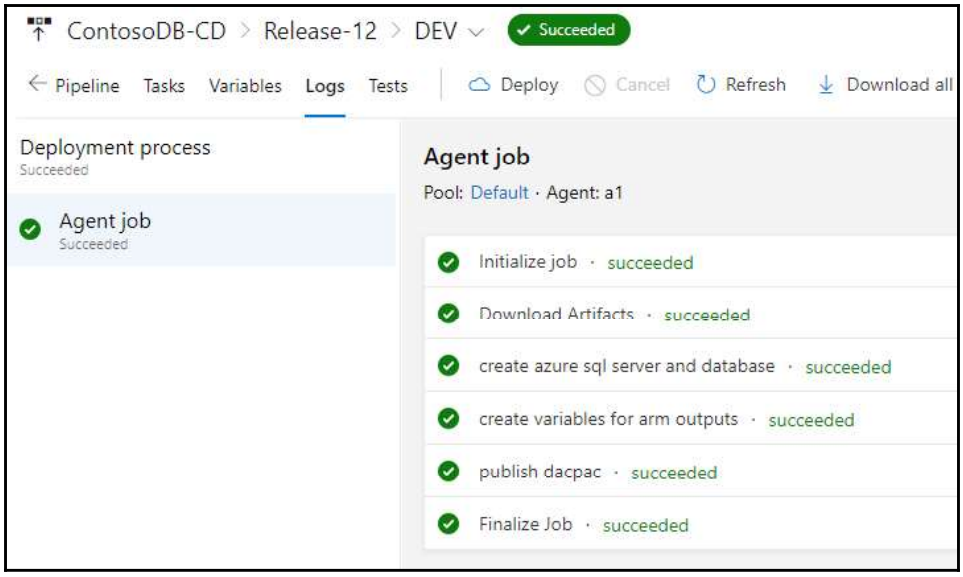
Creating the release pipeline

Now that we have the build pipeline ready and producing the database as an artifact, we are ready to consume it and deploy it to the environment.

1. Head over to the Release hub and create a new release pipeline. We will add two artifact sources to this pipeline. Add the artifact that was produced by our build pipeline:



8. Save the pipeline and create a release. Your release pipeline will now create all the required resources and deploy the `.dacpac` file:



How it works...

In this recipe, we saw how easy it is to provision Azure resources (SQL Server and database) and deploy them using Azure DevOps Server pipelines. We created a database project using Visual Studio, which allows us to maintain our database (schema and scripts) as code. We then created a build pipeline, which ensures that we are not introducing any breaking changes by continuously building changes into our database code. Lastly, we created the release pipeline and, using build artifacts and ARM templates, we provisioned the necessary resources and deployed the database.

In the next recipe, we will extend this pipeline to deploy to a new stage, named TEST, and see how we can make use of variable groups to consume secrets from Azure Key Vault.

Consuming secrets from Azure Key Vault in your release pipeline

This recipe is an extension of the previous recipe; if you haven't already read the previous recipe, I recommend that you read it first.

In the previous recipe, we saw how to keep strings, such as passwords as pipeline variables and how to mark them as secure variables so that they are not visible in the logs or to anyone else editing the pipeline once saved. While it works really well, enterprises that are deploying to the cloud would love to centrally manage and maintain these secrets in Azure Key Vault.

You can read more about Azure Key Vaults here: `http://bit.ly/2OAslff`.

Azure DevOps Server 2019 has native support for Azure Key Vault with variable groups. With variable groups in Azure DevOps Server, we can bring secrets from Azure Key Vault.

Getting ready

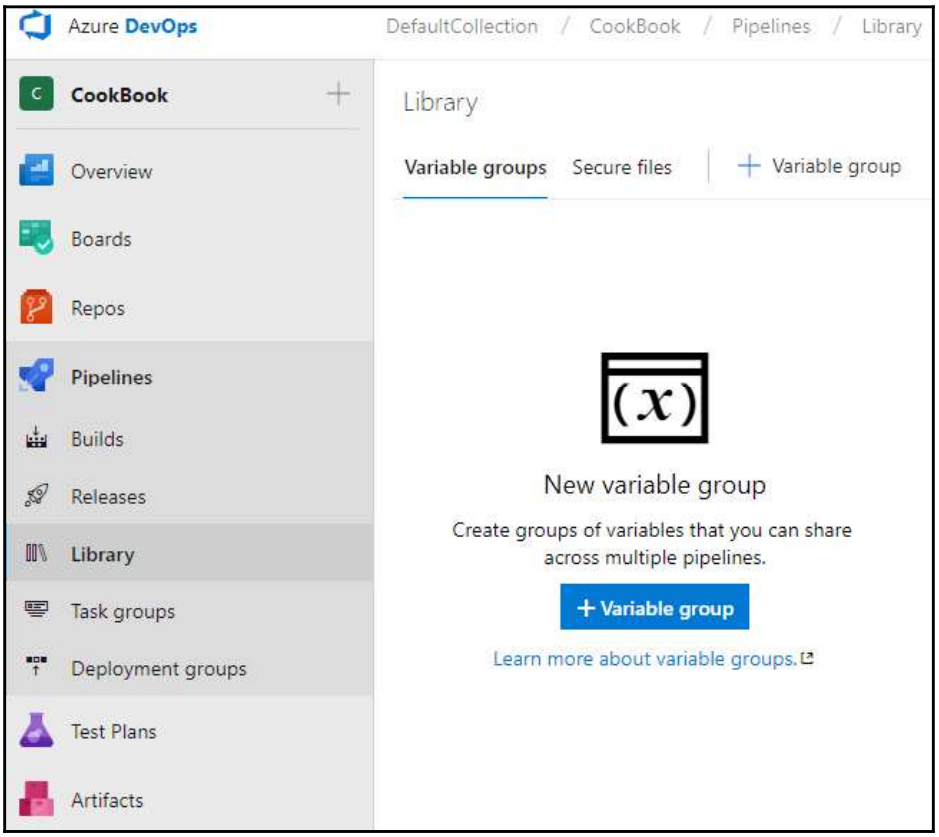
As a first step, we will manually create an Azure key vault and store the SQL Admin password as a single secret.

Creating a key vault in Azure

1. Go to `portal.azure.com` and then click on the **Create a resource** button. In the next blade, search for `Key Vault` and then click **Create**.

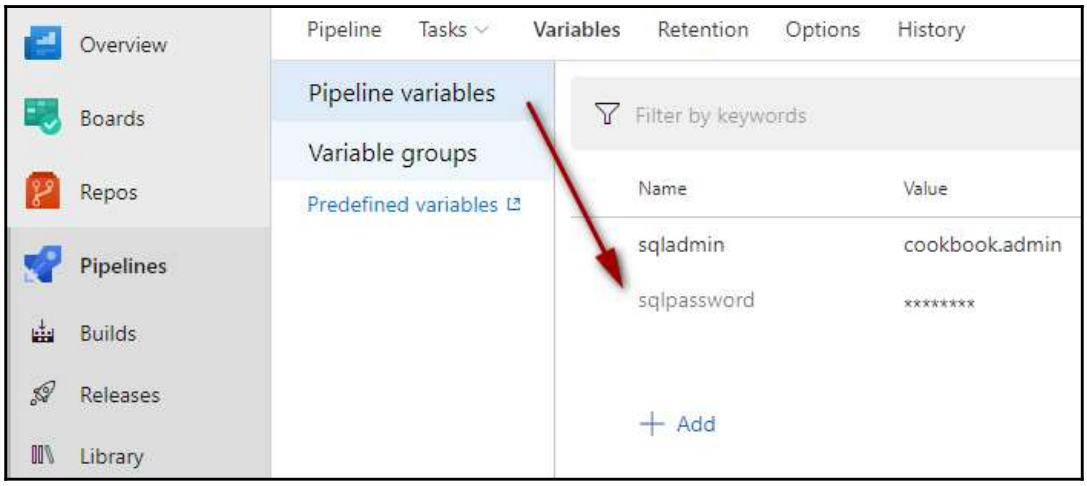
Creating a variable group and linking it to Azure Key Vault

Variable groups are defined and managed from the **Library** tab under the **Pipelines** tab. The advantage of a variable group is that you can make a set of variables available across the pipeline:



How to do it...

1. Go back to the release pipeline we created in the previous (*Deploying the database to Azure SQL using release pipeline*) recipe and enter the edit mode. Then, click the **Variables** tab. You might remember that we have a secret defined as a pipeline variable:



2. Because we will bring new value from the key vault via the variable group we defined earlier, we need to remove the pipeline variable, `sqlpassword`. Once done, click the **Variable groups** tab, then click the **Link variable group** button. You will see a new overlay window:

How it works...

Azure DevOps Server now intelligently brings in the latest values of the secret from the Azure key vault during runtime and passes them to the referenced task over the HTTPS channel. Each secret in the key vault is automatically created as a secret pipeline variable, which you can reference in our pipeline tasks like any other pipeline variables using the usual syntax (`%VARIABLE_NAME%` in a batch script, `$env:VARIABLE_NAME` in PowerShell, or `$VARIABLE_NAME` in bash scripts):

There's more...

Here are a few key facts about the variable group:

- Any changes to the secret values are automatically available during the run of the pipeline.
- Newly added secrets in the key vault are not automatically available in the pipeline. We will need to add them to the variable group.
- Deleting a variable group or removing the key-vault-linked secret from the variable will not remove the secret from the key vault.
- The variable group currently supports Azure key vault secrets only – cryptographic keys or the certificates are not supported.

See also

All the assets (variable groups and secure files) defined in the **Library** tab share the same security model. You can restrict who can create the variable group or the Library using permissions - For more on this is visit `http://bit.ly/2uPLWPp`.

Deploying the .NET Core web application to the Azure App Service

More and more users are switching to the .NET core framework these days. ASP.NET Core is a cross-platform framework for building modern applications. It offers many advantages over the traditional ASP.NET with many out-of-the-box features, such as dependency injection, which is suited for containers and those who want high performance.

For more on ASP.NET Core and its benefits, visit `http://bit.ly/2P0vMfn`.

In this recipe, we will create a simple ASP.NET Core web app and deploy it into Azure App Service.

Getting ready

Here, we will just use the `dotnet` command to create the basic ASP.NET core application and commit it into the git repository. Then, we will create a new build pipeline to build the application and produce the artifact.

Creating the ASP.NET Core application

1. Ensure that you have the latest .NET Core SDK installed. If not, install the recommended version from `https://dotnet.microsoft.com/download/dotnet-core`.

   You can see the installed SDKs on your machine by using the following command.

   ```
   c:\aspnetcore-demo>dotnet --list-sdks
   2.1.202 [C:\Program Files\dotnet\sdk]
   2.1.505 [C:\Program Files\dotnet\sdk]
   2.1.602 [C:\Program Files\dotnet\sdk]
   2.2.105 [C:\Program Files\dotnet\sdk]
   ```

   For this demo, I am using 2.2 .NET Core.

How to do it...

As we did in our first recipe, *Deploying the database to Azure SQL using the release pipeline*, we will also create the required infrastructure for our website using the release pipeline. To host our website in Azure, we need two things:

- **An app service plan**: Within Azure, an application runs inside the app service plan
- **An app service**: To host the website

- For more information on the app service plan, visit `http://bit.ly/2Pa9nwj`
- For more information on Azure App Service, visit `http://bit.ly/2PbfKzp`

Creating ARM templates

Let's create an ARM template that will create an app service plan and an empty app service:

1. Create a resource, `web.deploy.json`, and paste in the following content:

All the ARM templates referenced here are available under *RCP03-ASPNETCore-CD* folder

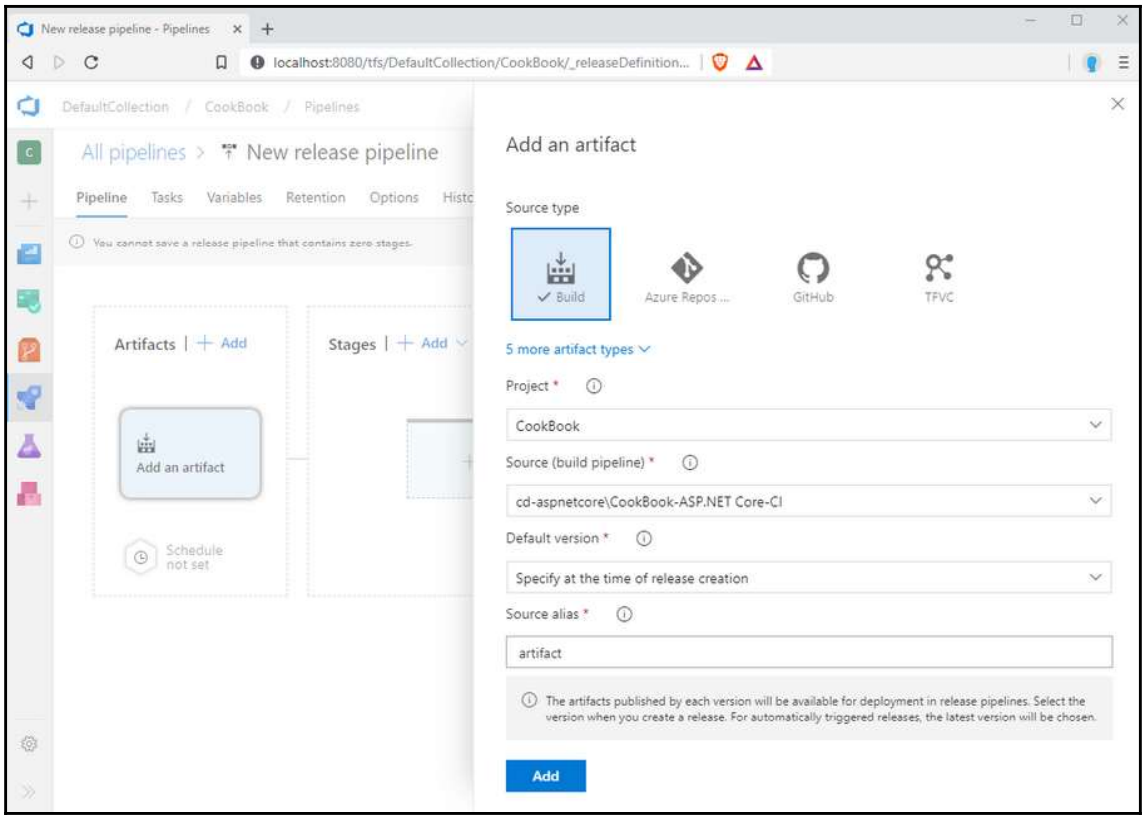
The important section is the `resources` array – you will see that in this ARM template, we are creating our app service plan and app service. We are also creating a slot named staging so that we can test our website before deploying it to the production slot (which is the default).

```
{
    "$schema":
"https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.
json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        //code is trimmed for the sake of brevity
    },
    "variables": {
        "webAppServicePlanName":
"[concat(parameters('environmentConfiguration').prefix.appServiceWeb)]",
```

Creating the release pipeline

Let's start building the release pipeline to deploy the application:

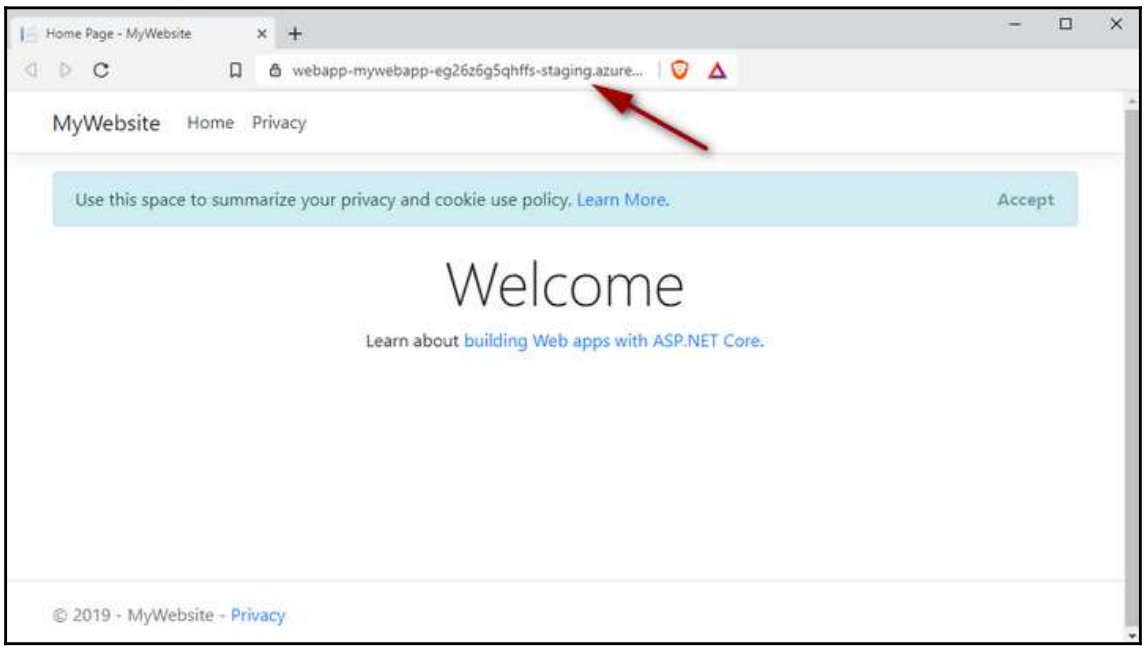
1. The first step is to create a new release pipeline and add the build artifact. To do that, Go to the *release* hub and create a New Release Pipeline and add the artifact.



As you can see from the preceding screenshot, we are linking the build pipeline, which produces the deployable artifact for this release pipeline. Just for demonstration purposes, we are also setting the default version as **Specify at the time of release creation**, which means during the release creation, we will have to select the version of the artifact to be deployed we want to use.

How it works...

Once you create the release, the deployment will start. The first step in the pipeline will first create an App Service Plan and other resources (slots for example) specified in the ARM template. The next step is to create the pipeline variables using the ARM Outputs task so that we can access the app service name we created in the previous task. Finally, we are using the **Azure App Service Deployment** task to deploy the application to the staging slot. We can then browse to the staging slot to see our website up and running:



There's more...

In this recipe, we saw how we could deploy a web application from no infrastructure to a fully-hosted website in Azure using Azure DevOps Server 2019. We also saw how deployment slots allow us to isolate our deployments. We could extend this pipeline with more stages, such as DEV, TEST, and PROD. We can then use slots to isolate and open the new version of the website to only specific teams (say testing team) before swapping it with the production slot.

More information about deployment slots and management can be found at `https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots`.

See also

Check out the following resources:

- Considerations on using Deployment Slots in your DevOps pipeline: `http://bit.ly/2P95vM5`
- App Service Plans: `http://bit.ly/2P905R1`

Deploying an Azure Function to Azure

Azure Functions are a new way to run your logic on a **serverless** technology in the cloud. Azure functions are hugely popular mainly because they can be cheaper compared to app service - as you have an option to pay only for the time spent running your code.

- Azure Functions documentation: `http://bit.ly/2Pb36jP`
- Serverless in Azure: `https://azure.microsoft.com/en-us/solutions/serverless/`

In this recipe, we will look at how to create an Azure Function in TypeScript and then we will look at how to deploy a sample Azure Function to Azure.

Getting ready

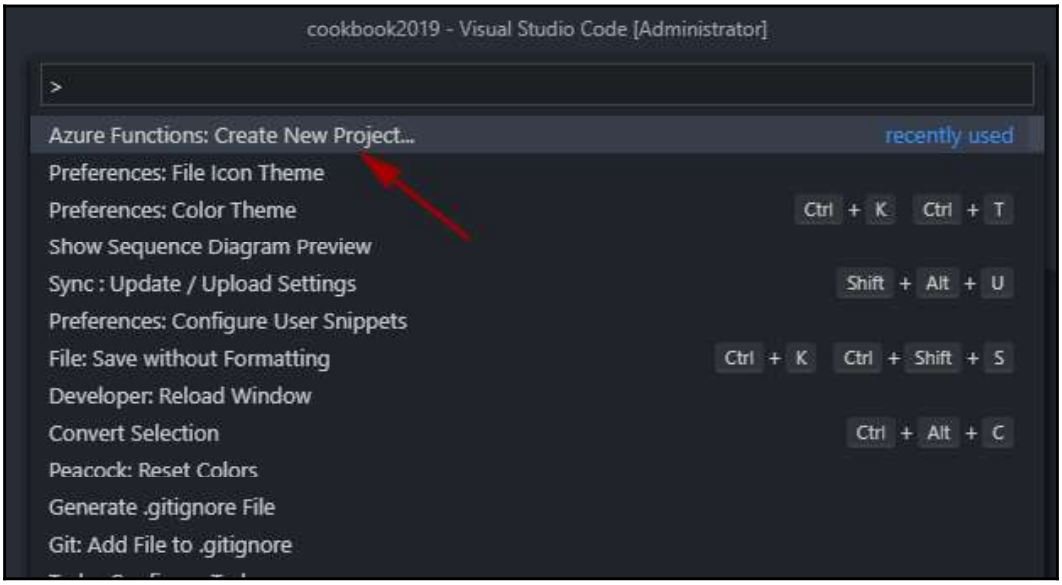
To create the Azure Function, you will need the following tools installed on your machine. Go ahead and install them all.

- VSCode: `https://code.visualstudio.com/`
- Azure Functions Extension for VS Code: `http://bit.ly/2Pd9OGk`
- NodeJS 8.0 and above: `https://nodejs.org/en/`
- PostMan: `https://www.getpostman.com/`

Creating a sample Azure Function

Once you install all the tools mentioned in the Getting ready section above,

1. Open Visual Studio Code, press *F1*, select **Azure Functions: Create New Project**, and then select **Browse** and select a folder to create the required files using the wizard:



The source code for Azure Function is in the GitHub repository under `RCP04-AzureFunction-CD` folder.

Creating the build pipeline

Creating the build pipeline is similar to what we did previously in *Deploying the database to Azure SQL using the release pipeline* recipe. This is a typescript project, and we have a few scripts in our scripts section in our `package.json`. The following is the YAML file for our build pipeline. As you can see, this is just made up of three tasks:

- Install the dependencies
- Build the project
- Publish the artifacts

These 3 steps are under *steps* section in the below YAML content.

The YAML file is under relevant chapter folder under *RCP04-AzureFunctions-CD* directory in the code bundle.

```
resources:
- repo: self
queue:
  name: Default
  demands: npm

trigger: none

steps:
- task: Npm@1
  displayName: 'npm install'
  inputs:
    workingDir: '$(build.sourcesdirectory)/continuous-deployments/rcp-
deploy-az-function/Function'
    verbose: false

- task: Npm@1
  displayName: 'install func cli'
  inputs:
    command: custom
    workingDir: '$(build.sourcesdirectory)/continuous-deployments/rcp-
```

How to do it...

Now that our artifacts are all exactly how we want them to be, we can start building the release pipeline. Like we did in the previous recipes, we need to create the Azure Function app using ARM template and then deploy our function code in to the created Azure Function app. For this, we need to create an ARM template to provision the Azure Function app. As we mentioned previously, the Azure Functions provide cost benefits (you pay only for the time your code runs) over traditional websites, which require a full web server. For this purpose, Azure Functions has two kinds of pricing plans (which you can learn more about at `https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview`):

- **Consumption plan**: When your function runs, Azure provides all of the necessary computational resources. You don't have to worry about resource management, and you only pay for the time that your code runs.
- **App Service plan**: Run your functions just like your web apps. When you are already using App Service for your other applications, you can run your functions on the same plan at no additional cost.

For this example, we are going to use the Consumption plan to ensure that we pay only for the time our function runs.

Creating the ARM template

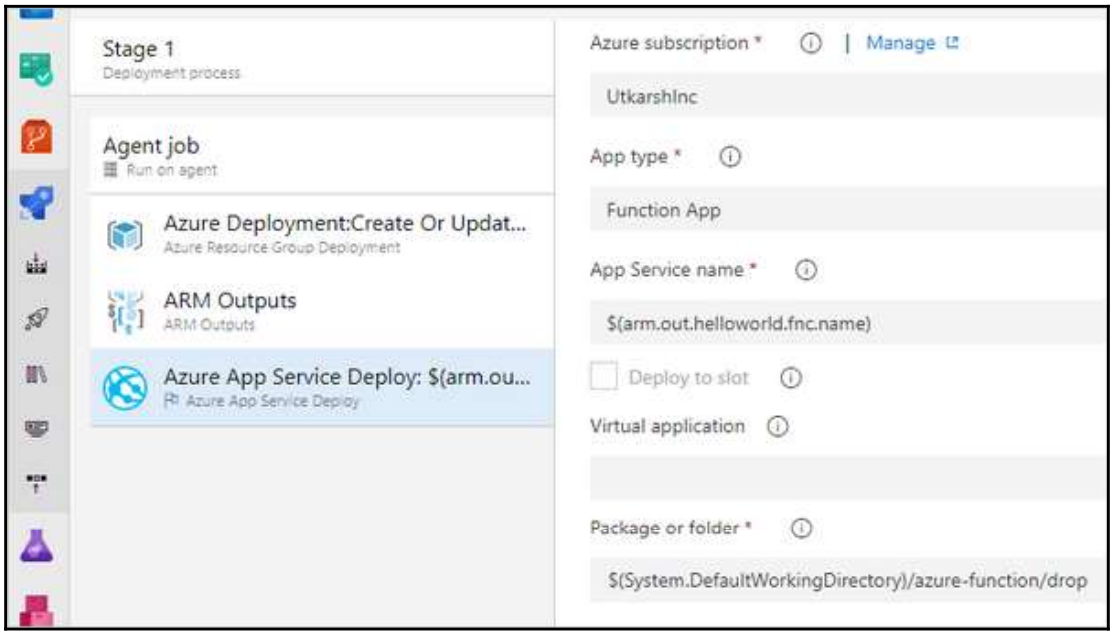
1. Create a JSON file called `function.deploy.json` and copy the contents from the ARM template provided in the code bundle.

The complete ARM template used in the recipe is available in the code bundle under *RCP04-AzureFunction-CD* folder with file named function.deploy.json

Notice the *resources* section. The first resource is the app service plan. We are setting `computeMode` as Dynamic, this ensures we are using the consumption plan. The second resource creates an Azure Function app resource.

```
"resources": [
       {
           "type": "Microsoft.Web/serverfarms",
           "apiVersion": "2015-04-01",
           "name": "[variables('appAppServicePlanName')]",
           "location": "[resourceGroup().location]",
```

4. Publish the function package to the created Azure Function application:



How it works...

In this recipe, we saw how to build a simple Azure Function and produce the artifacts in the build pipeline. We then saw how to create the Azure Function application using ARM templates. The release pipeline creates the required resources in Azure and then deploys our function into the provisioned function app. If you go to the portal and browse the resource group, you will see three resources created:

See also

Here are some helpful links regarding what we covered in this recipe:

- Automating resource deployment for your function app in Azure Functions: `http://bit.ly/2v93DcU`
- Durable functions: `http://bit.ly/2vakoo1`
- Provisioning a function app on a Consumption plan (ARM templates): `http://bit.ly/2v7f1pN`
- Creating serverless applications: `http://bit.ly/2v5wD5t`

Publishing secrets to Azure Key Vault

Applications contain many secrets, such as connection strings, passwords, certificates, and tokens, which, if leaked to unauthorized users, can lead to a severe security breach. This can also result in serious damage to the reputation of the organization and can cause compliance issues.

Azure Key Vault allows you to manage your organization's secrets and certificates in a centralized repository. The secrets and keys are further protected by **Hardware Security Modules** (**HSMs**). It also provides versioning of secrets, full traceability, and efficient permission management with access policies.

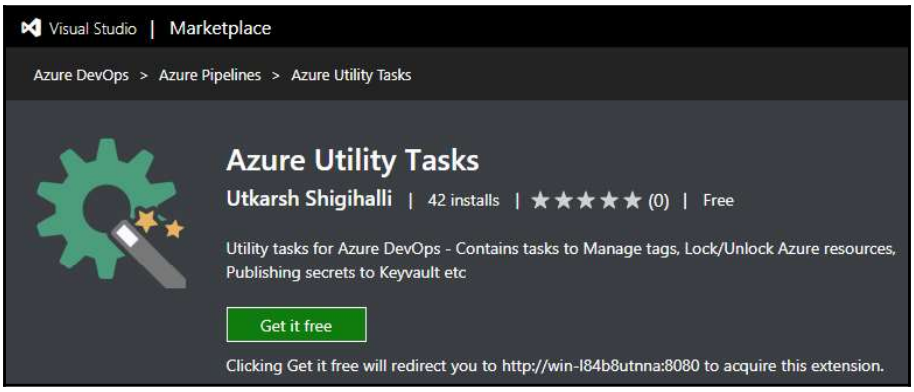
For more information on Azure Key Vault, visit `https://docs.microsoft.com/en-us/azure/key-vault/key-vault-overview`.

In this recipe, we will see how we can automatically publish secrets in our pipeline so that secret management is automated.

Getting ready

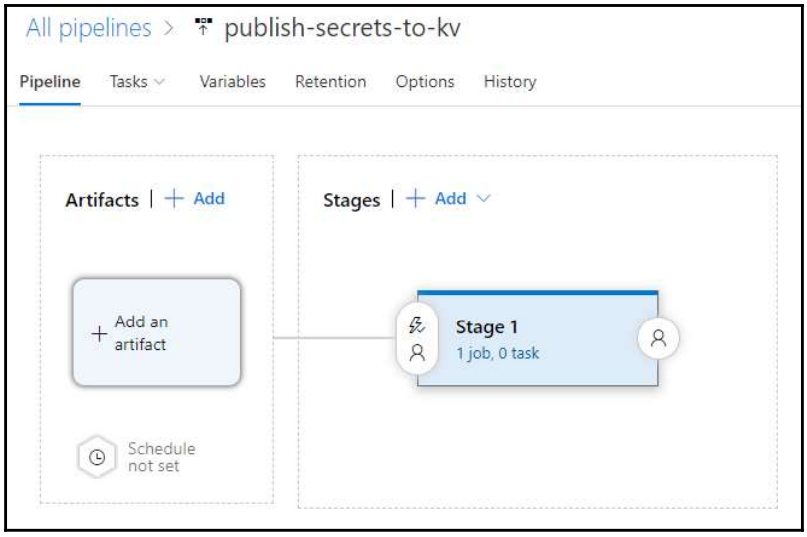
For this recipe, we are assuming you already have a key vault in the Azure portal. If you don't, please refer to the *Creating a key vault in Azure* section in the *Consuming secrets from Azure Key Vault in your release pipeline* recipe.

Next, install a marketplace extension named Azure Utility Tasks (`http://bit.ly/2PiPQtJ`). This extension provides a few utility tasks, and one of them publishes secrets to Azure Key Vault. We will see how we can use this task shortly:

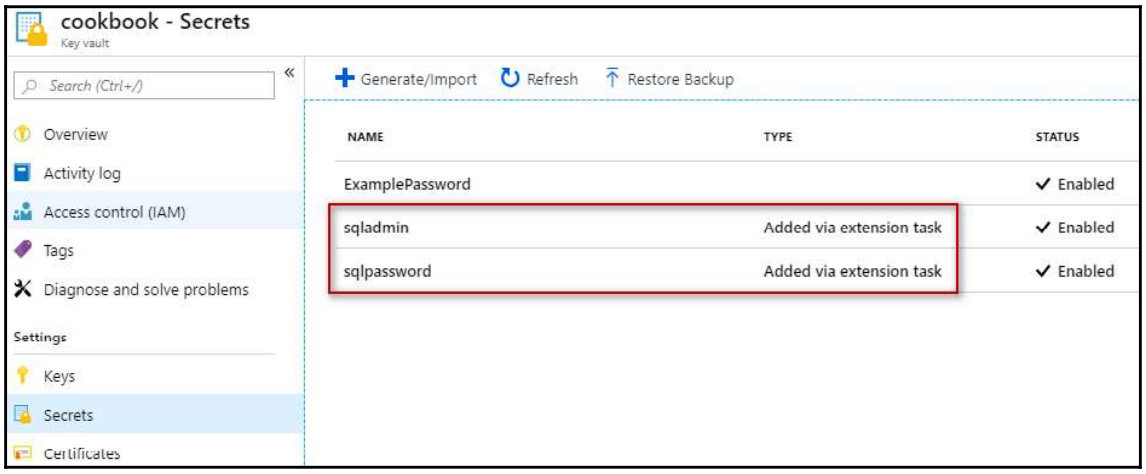


How to do it...

1. Let's create the release pipeline, add a stage, and save it. This is similar to what we have done in other recipes in this chapter. Assuming you have a stage with no artifacts, your release pipeline will look as follows:



As you can see from the preceding screenshot, the task also allows you to select the Key Vault from the dropdown, and each secret is separated by a new line. Run the release and you will see all your secrets in the key vault being added:



How it works...

The automation of publishing secrets to the key vault greatly reduces your dependency on manual scripts and also any errors in doing so. In this recipe, you saw how easy it was to insert secrets into the azure key vault, which is a central repository to manage all your secrets, keys, and certificates. The recipe showed you two ways that you can automate inserting secrets into the key vault via the Azure DevOps Server release pipeline.

There's more...

We could extend this pipeline to provision the key vault itself. The steps would be similar to what we have done in the previous recipes. A step to deploy the ARM template which will use a Resource Group Deployment task to provision the key vault and a step to get the provisioned key vault name as an output parameter and eventually using it to add secrets to the provisioned key vault.

In doing so, we can extend this pipeline to automate end-to-end key-vault provisioning and also inserting secrets after creation.

See also

Check out the following resources to learn more about what was covered in this recipe:

- Azure Key Vault ARM templates: `https://docs.microsoft.com/en-gb/azure/templates/microsoft.keyvault/allversions`
- Azure Key Vault best practices: `https://docs.microsoft.com/en-gb/azure/key-vault/key-vault-best-practices`

Deploying a static website on Azure Storage

Static websites have become very popular in the last few years and are based on the JAMstack (JavaScript, APIs, and Markup) architecture. The generated websites are super lightweight, fast, and easier to develop. As of December 2018, you can host static websites on Azure Storage accounts of the **General Purpose v2** (**GPv2**) type.

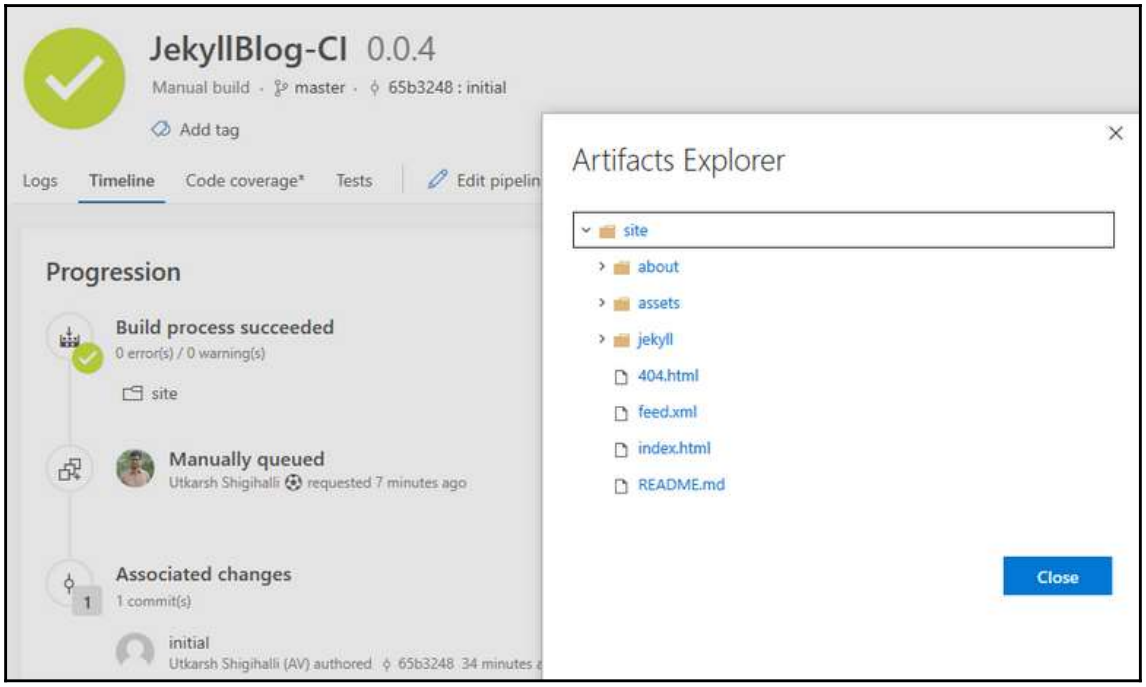
In this recipe, we will see how we can configure a storage account to host a static website. We will then deploy a simple static website to this storage account so that we can browse our website.

- For more information on what JAMStack architecture - `https://jamstack.org/`
- More information on Static website hosting on Azure Storage: `http://bit.ly/2PlYtne`

Getting ready

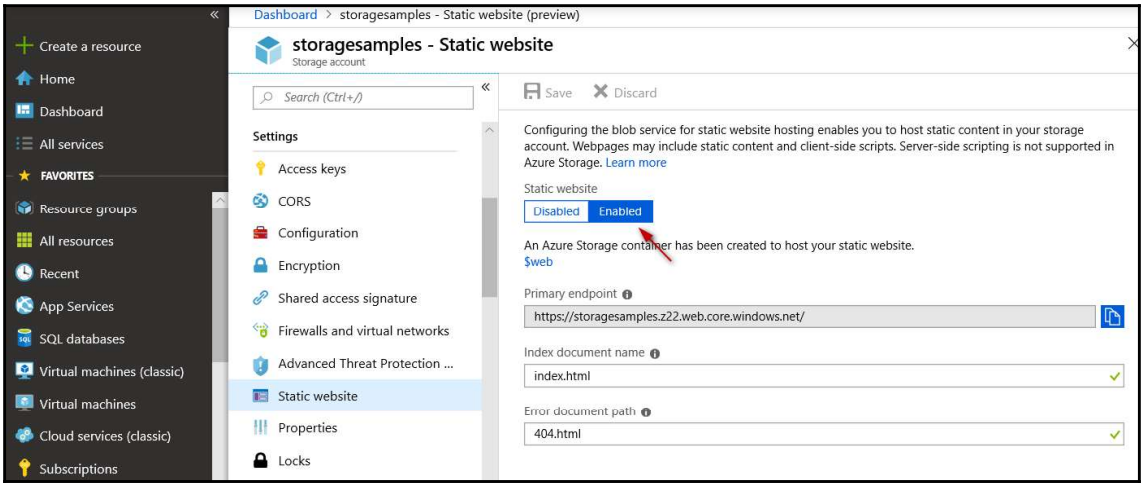
To host the website, we will first need to have a static website. Creating a static website using JAMStack architecure is outside the scope of this recipe, but you can check out this post on how to build and automate publishing a Jekyll website: `http://bit.ly/2PlRcDU`.

I am assuming you already have a static website and published artifact, as follows:



Creating a storage account from the Azure portal

From the Azure portal, it is to create a storage account of the **General Purpose v2** (**GPv2**) type. You will see a **Static website** setting - Enable the setting and optionally set the index document name and error document name. Once static website hosting is enabled, a container named `$web` will be created, if it doesn't already exist. Any content copied to the `$web` container will automatically be served on the primary endpoint:



Files on the `$web` container are served through **anonymous** access requests and will only have read permissions.

Creating an Azure Storage Account ARM templates

Automating the creation of the GPv2 storage account and enabling this setting will require a bit more work. Let's start by creating an ARM template.

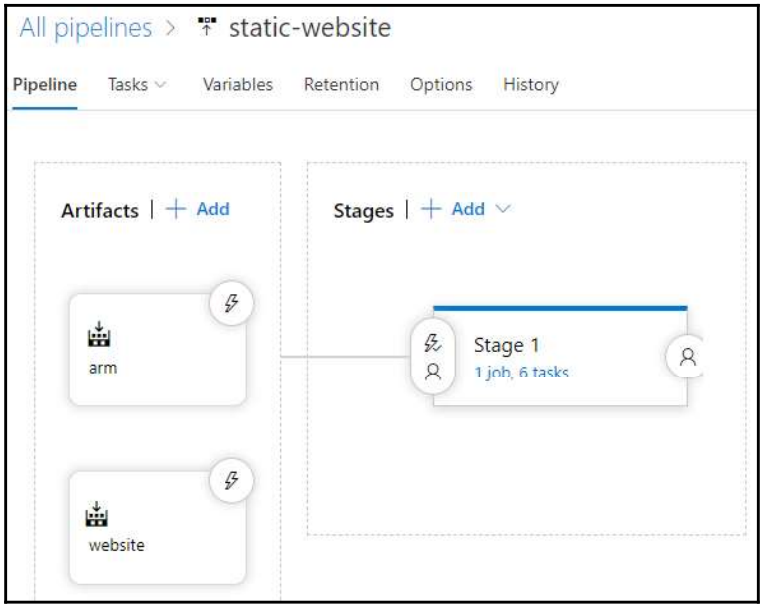
Create a JSON file named `storageaccount.deploy.json` and paste in the following content:

The source code is available under Chapter folder inside RCP06-StaticWebsite-CD directory.

```
{
    "$schema":
"http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.j
```

How to do it...

1. Create a new release pipeline and add both the static website and ARM templates as artifacts:



The first steps will just deploy our storage account ARM template. These steps are similar to what we have done in other recipes. The first task deploys the ARM template and the next task produces the pipeline variables for our ARM template output variables. In our case, we will output the storage account name and the storage account primary endpoint from our ARM template:

2. Run the release. We see that our storage account has been created, but that the static website setting is still disabled:

How it works...

The pipeline we created in this recipe shows how we can use ARM templates and Azure CLI commands to easily automate the creation of required storage account to deploy a static website. We saw how, even when the ARM template does not provide full capabilities to automate the *Static Website* feature, the Azure DevOps server helps us to integrate any tool available into the pipeline—in this case, we used the Azure CLI to enable the static website feature and copy its contents.

There's more...

Hosting a static website on Azure Storage makes your site available on the primary endpoint. However, in most scenarios, you would like to host your website on your custom domain, such as `https://www.myorganization.com/blog`. You could do that using the Azure **Content Delivery Network** (**CDN**). Azure CDN also allows you to use custom SSL certificates, rewrite rules, and more.

For more on how to use Azure CDN and enabling custom domains for your static website, visit `http://bit.ly/2vgXvQ8`.

See also

Check out these resources for more information:

- We used Jekyll to generate a static website, but it is just one of the many static generators that's available. There is a full list of static site generators here: `https://www.staticgen.com/`.
- You can configure a custom domain name for your Azure storage account at `http://bit.ly/2venQ1b`.

Deploying an Azure Virtual Machine to Azure Dev Test Lab (DTL)

Development teams are often limited by the infrastructure that is available to them to deploy and test their changes. The cloud promises to address this by giving you an infinite resource capacity that you can consume in a pay-as-you-go subscription model. Enterprises making their first foray into the cloud are keen to test the waters by moving development and test workloads to the cloud. However, the biggest apprehension when moving to the cloud for Development and Testing teams is repeatability, security, and governance. Microsoft understands the trend, so to help customers make the move, it has introduced a new service called Azure Dev Test Lab.

Azure DTL is a service that helps development teams quickly create heterogeneous environments in Azure while minimizing waste and controlling cost. The biggest unique selling proposition for Azure DTL is the ability to lock down the lab by securing the network to a private subnet, applying governance policies at the lab level, and giving the development teams autonomy within the lab. The ability to create and repeat helps scale the solutions and the integration with the existing toolchain helps reusability.

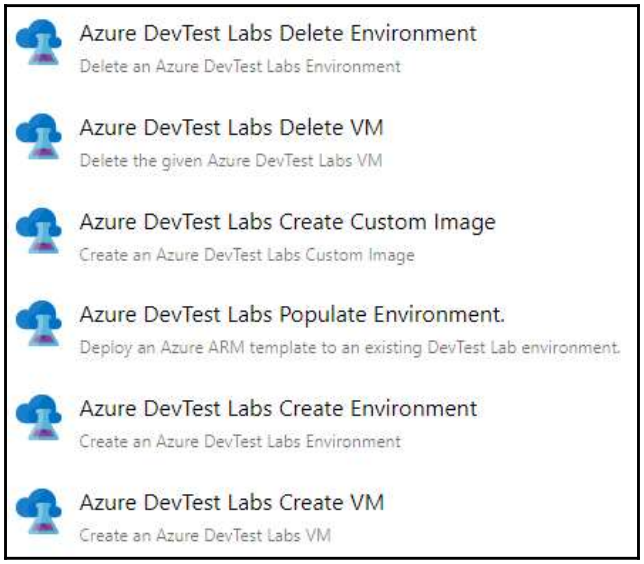
You can learn more about Azure DTL by watching this introductory video: `https://azure.microsoft.com/en-gb/resources/videos/index/?services=devtest-lab`.

If you don't already have an Azure DTL in your Azure subscription, you can create one by following this walkthrough: `https://docs.microsoft.com/en-gb/azure/lab-services/tutorial-create-custom-lab`.

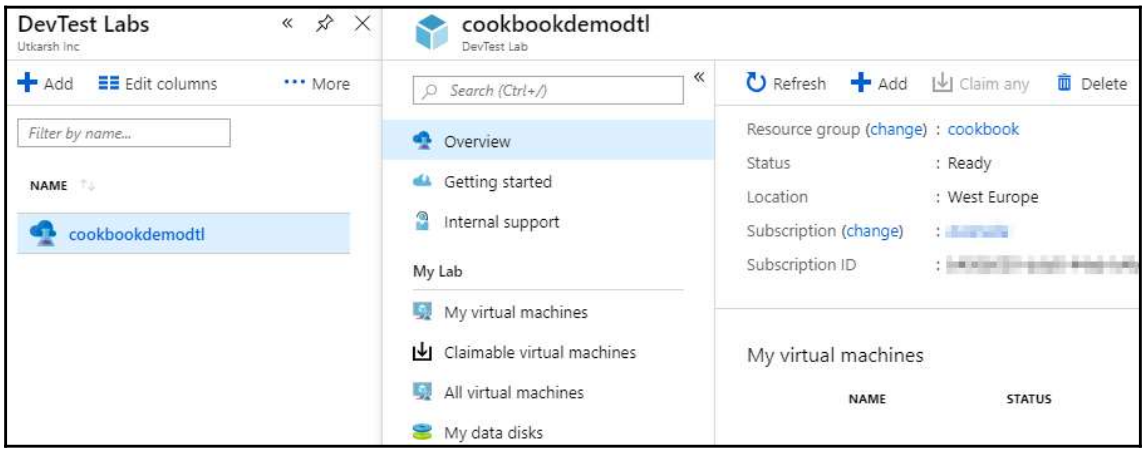
In this recipe, we'll learn how to securely connect our Azure DevOps server to an Azure subscription. We will then use Azure DevOps Server to provision virtual machines using an ARM template into the newly created Azure Dev Test Lab.

Getting ready

The Azure Dev Test Labs team provides a free Visual Studio marketplace extension. This free extension provided by Microsoft delivers multiple builds and release pipeline tasks that allow you to create machines, delete machines, and create custom images from existing machines in Azure DTL:

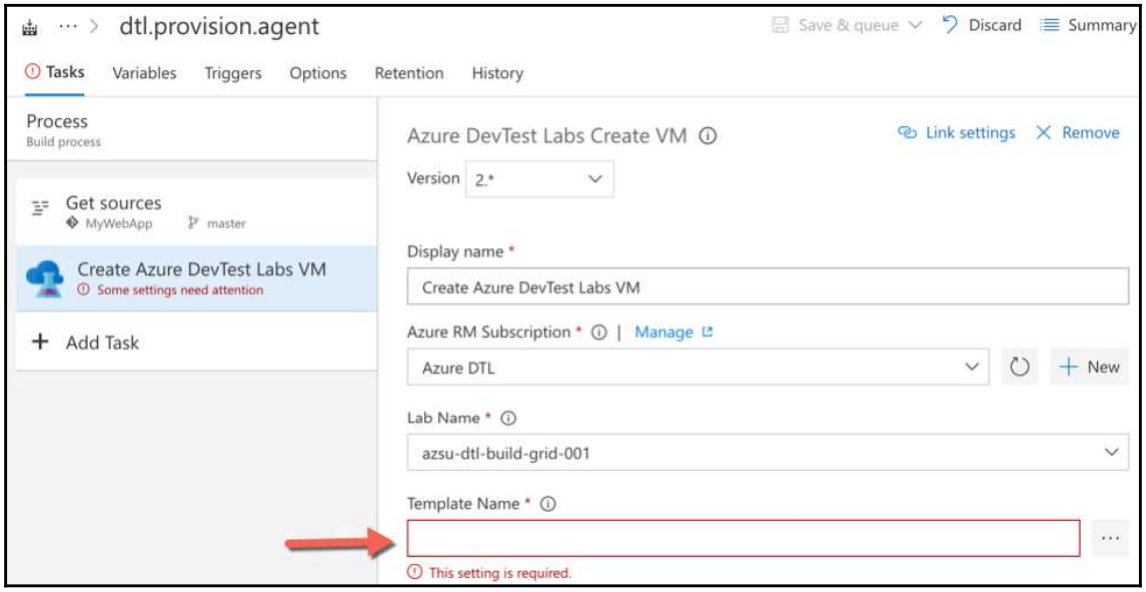


For simplicity, we also have an Azure DTL ready that was manually created using the portal:



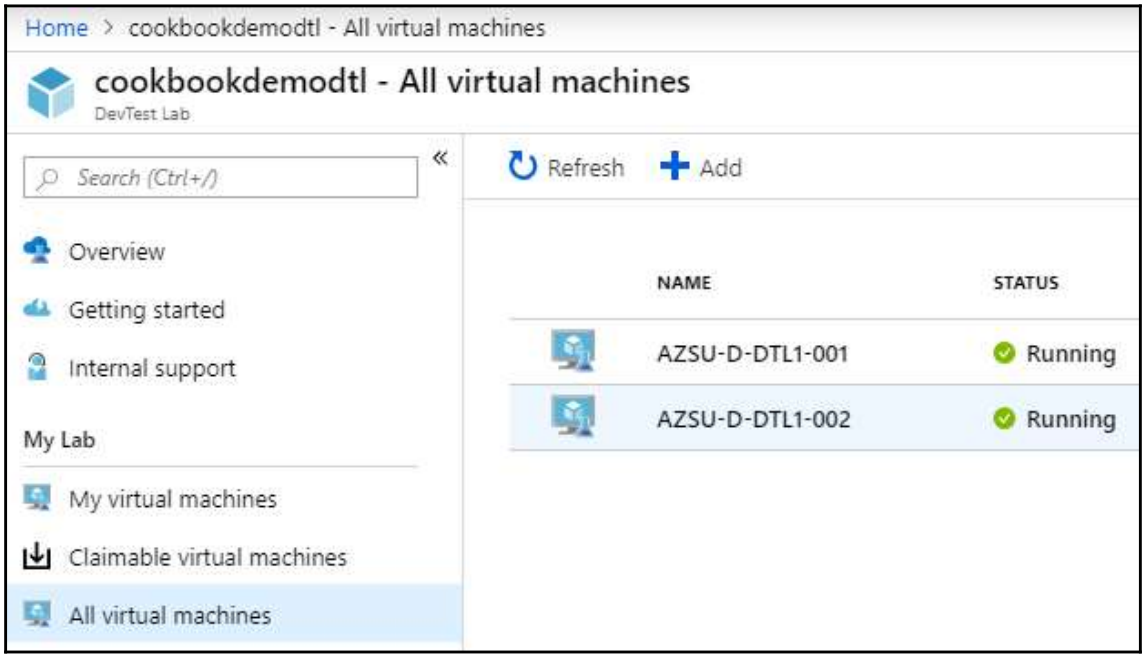
How to do it...

1. Create a release pipeline and add the Azure DevTest Labs create VM task. Select the subscription and provide the ARM template:



For the template field, you could provide the ARM template for the custom VM image according to your needs. You'll be delighted to know that it's possible to generate an ARM template for provisioning right from within the Azure portal.

5. Run the release pipeline and you should soon see a new VM spun up based on the ARM template and added to the DevTest labs:

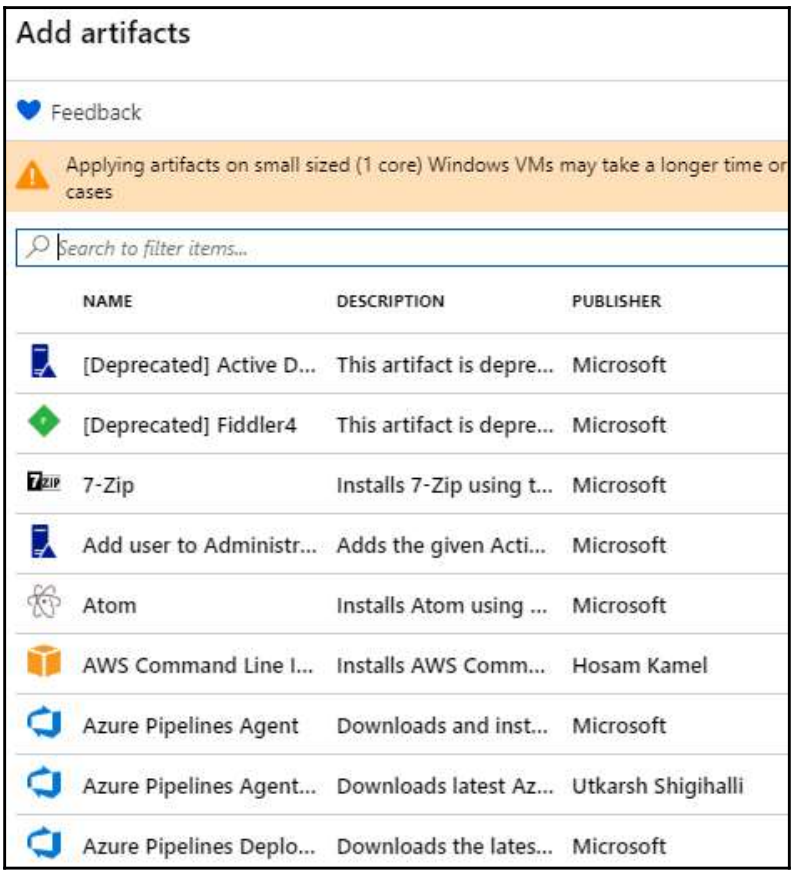


How it works...

Thanks to the power of Azure DevOps Server and ARM templates, we saw how we can generate ARM templates for our custom DevTest Labs VM images. We then used the generated ARM templates to spin new VMs in our Azure DevTest Labs lab.

There's more...

We could extend this recipe, for example, to build an Azure DevOps Server build agent grid. An automated process to add and remove build agents allows you to scale up and scale down on demand. There will always be periods when the build infrastructure is in high demand and periods when it's underutilized. By using virtual infrastructure to host your agents, you could save significant money by decommissioning the agents when they are not in use. This recipe showed you a quick way to spin the VMs on demand. We could add artifacts that are available for DTL VMs (the Azure Pipelines Agent artifact, for example) and generate an ARM template with it to automatically create a VM and add an artifact:



See also

DevTestLabs Artifacts allow you to add the custom software/tools you need to your Azure DTL VMs as you provide them. You are not limited to using just the available artifacts. You could build your own custom artifacts, which is very easy to do. Check out these resources for more information:

- Create custom artifacts for your DevTest Labs virtual machine: `http://bit.ly/2vi2akG`
- All of the Azure DTL artifacts are open source on GitHub, which you could use as references: `https://github.com/Azure/azure-devtestlab/tree/master/Artifacts`