# Building an Azure DevOps Pipeline for Infrastructure

When you search online, you will find various blog posts, documentation, and tutorials on Azure DevOps. These items are valuable resources but rarely does one walk you through a real-world scenario. Many skim over the security aspect leaving passwords in clear text for simplicity or an end product that essentially does nothing. Let's change that.

In this project, you'll learn from soup to nuts how to build a *real* Azure DevOps release pipeline that automates infrastructure. You'll learn to use Azure DevOps to build a continuous deployment pipeline to provision Azure virtual machines.

## Project Summary

By the end of this project, you will have a fully-functioning Azure pipeline. From a single GitHub repo commit, it will:

- Build a temporary Azure resource group
- Provision an Azure VM via an ARM template
- Set up said ARM template in a CI/CD pipeline
- Upon any change to the template, kick off a template validation test
- Deploy the ARM template to Azure
- Test the deployed infrastructure
- Tear down all Azure resources

Let's dive right in!

## Project Overview

This project is going to be broken down into six main sections. They are:

### Azure resource preparation

In this section, you will learn how to set up all of the prerequisite resources in Azure. Here you will:

- Create an Azure service principal for various tasks in the pipeline
- Set up an Azure Key Vault with secrets for the pipeline to use
- Set appropriate access policies for ARM deployments and pipeline usage

## Azure DevOps preparation

Once you have all of the Azure resources set up, it's time to prepare Azure DevOps for your pipeline. In this section, you will:

- Install the Pester Test Runner build task in your Azure DevOps organization
- Create service connections to provide Azure DevOps with required resource access
- Create an Azure DevOps variable group linking a key vault to access Azure Key Vault secrets

## Script/template overview

Various artifacts go with this project, including the ARM template to build the server and the Pester tests. In this section, we'll briefly cover what the template is provisioning and what exactly Pester is testing in the pipeline.

## Pipeline creation

This section is where the real fun begins. You will begin setting up the actual pipeline. You'll learn to set up this entire orchestration via a single YAML file.

> 💡 *You'll be building the pipeline using the Multi-Stage Pipeline UI experience. As of this writing, this feature is in Preview.*

## Pipeline demonstration

Once the pipeline is built, you need to see it run! You'll learn how to trigger the pipeline and watch the magic happen in this section.

## Cleanup

And finally, since this is just a demonstration, you'll get access to a script to tear down everything built during the Project.

Does this sound like a lot? It is! But don't worry; you'll learn step-by-step as attack each task one at a time.

> 💡 *If you'd like a script with all of the Azure CLI commands used to build this pipeline, you can find it in the ServerAutomationDemo GitHub repo as demo.ps1.*

## Prerequisites

You'll learn a lot, but you're also expected to come to the table with a few things. If you plan to follow along, be sure you have the following:

- An Azure DevOps organization - Check out the Microsoft QuickStart guide for instructions on how to do this. In this Project, you'll be working on a project called *ServerAutomationDemo*.
- A GitHub repo - In this project, you'll learn from a GitHub repo called *ServerAutomationDemo*. Sorry, we're not using Azure repos in this Project.
- A GitHub personal access token - *Be sure to create the token with the permissions of admin:repo_hook, all repo, and all user options.*

## New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

**Note**

AzureDevOps

What's this token for?

**Select scopes**

Scopes define the access for personal tokens. Read more about OAuth scopes.

| | |
|---|---|
| **repo** | Full control of private repositories |
| repo:status | Access commit status |
| repo_deployment | Access deployment status |
| public_repo | Access public repositories |
| repo:invite | Access repository invitations |
| **write:packages** | Upload packages to github package registry |
| **read:packages** | Download packages from github package registry |
| **delete:packages** | Delete packages from github package registry |
| **admin:org** | Full control of orgs and teams, read and write org projects |
| write:org | Read and write org and team membership, read and write org projects |
| read:org | Read org and team membership, read org projects |
| **admin:public_key** | Full control of user public keys |
| write:public_key | Write user public keys |
| read:public_key | Read user public keys |
| **admin:repo_hook** | Full control of repository hooks |
| write:repo_hook | Write repository hooks |
| read:repo_hook | Read repository hooks |
| **admin:org_hook** | Full control of organization hooks |
| **gist** | Create gists |
| **notifications** | Access notifications |
| **user** | Update all user data |
| read:user | Read all user profile data |
| user:email | Access user email addresses (read-only) |
| user:follow | Follow and unfollow users |

ARM deployments allowed access to the key vault

- Cloud Shell or PowerShell 6+ if running locally - Examples may work in Windows PowerShell but were not tested. All of the examples will be performed locally via a PowerShell console, but the Cloud Shell will work just as well. You will automate building the pipeline.
- Azure CLI installed (if running locally) - You'll learn how to perform tasks in this Project with the Azure CLI. But, the same procedures can also be performed via the Azure Portal, PowerShell or the Azure SDK.

> 💡 *Warning: The actions you're about to perform cost real money unless you have some Azure credit. The most cost-intensive resource you'll be bringing up in Azure is a VM but only temporarily.*

## Before you Start

You're going to be doing a lot of configuration in this Project. Before you begin, please be sure you have the following items handy.

- The name of the Azure subscription resources will be deployed to - the examples will use *Adam the Automator*.
- The ID of the subscription
- The Azure AD tenant ID
- The DevOps organization name - the examples will use *adbertram*.
- The region you're placing resources into - the examples will use *eastus*.
- The name of the Azure resource group to put the temporary key vault into - the examples will use *ServerAutomationDemo*.
- A password to assign to the local administrator account on a deployed VM - the examples will use "*I like azure.*".
- The URL to the GitHub repo - the examples will use *https://github.com/adbertram/ServerAutomationDemo*.

## Logging in with the Azure CLI

You should be ready to do much work with the Azure CLI in the Project. Although the Azure PowerShell cmdlets do a lot, the Azure CLI can do more DevOps tasks.

Your first task is getting to a PowerShell 6+ console. Once in the console, authenticate to Azure using the command `az login`. This command will open a browser window and prompt you for your account.

Once you've been authenticated, please set your subscription to the default. Setting it as the default will prevent you from having to specify it constantly.

```
az login
az account set --subscription 'Adam the Automator'
```

# Preparing Azure Resources

Once you're logged in with the Azure CLI, it's time to get to business. An Azure Pipeline has many different dependencies and various knobs to turn. In this first section, you're going to learn how to do some setup and prepare your environment for your pipeline.

## Installing the Azure CLI DevOps Extension

You'll need a way to build the various Azure DevOps components with the Azure CLI. By default, it doesn't include that functionality. You'll need to install the DevOps extension to manage Azure DevOps from the Azure CLI.

Luckily, installing the extension is a single line as shown below.

```
az extension add --name azure-devops
```

Once the extension has been installed, set your organization as the default to prevent specifying it over and over.

```
az devops configure --defaults organization=https://dev.azure.com/adbertram
```

## Creating the Resource Group

Although the pipeline will create a temporary resource group, you should create one for any resources brought up in this demo. This resource group is where you'll create an

Azure Key Vault.

```
az group create --location "eastus" --name "ServerAutomationDemo"
```

## Creating the Azure Service Principal

The next task is to create an Azure service principal. You will need an Azure service principal to authenticate to the Azure Key Vault. You'll also use this service principal to authenticate a service connection. Create the service principal for both the key vault and for the eventual ARM deployment, as shown below.

```
$spIdUri = "http://ServerAutomationDemo"
$sp = az ad sp create-for-rbac --name $spIdUri | ConvertFrom-Json
```

At this point, it'd be a good idea to save the value of $sp.appId somewhere. When you get to building the pipeline later, you will need this!

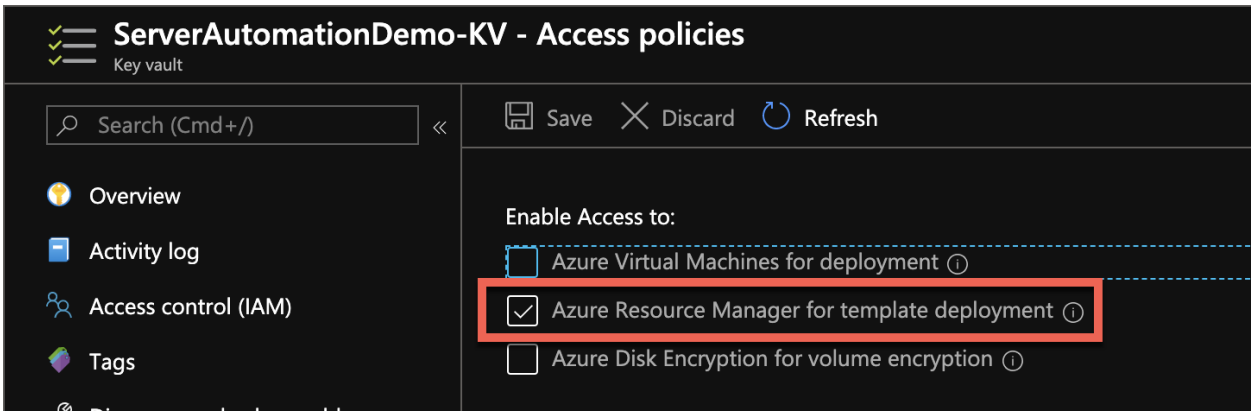
*You'll notice some PowerShell commands in the examples e.g.* `ConvertFrom-Json`*. Since the Azure CLI only returns JSON strings, it's easier to reference properties if converted to a PowerShell object.*

## Building the Key Vault

The pipeline in this Project needs to reference a couple of passwords. Rather than storing passwords in clear text, let's do it correctly. All sensitive information will be stored in an Azure Key Vault.

To create the key vault, use the `az keyvault create` command as shown below. This command creates the key vault in the previously-created resource group. Also, notice the `enabled-for-template-deployment` switch. This changes the key vault access policy to allow the future ARM deployment to access the key vault.

```
az keyvault create --location $region --name "ServerAutomationDemo-KV" --resource-group "S
erverAutomationDemo" --enabled-for-template-deployment true
```




Allowing ARM to access the keyvault

## Creating Key Vault Secrets

Once the key vault is created, it's time to create the secrets. For this demo, create two secrets called *ServerAutomationDemo-AppPw* and *StandardVmAdminPassword*. The *AppPw* password is the password for the service principal. The VM password will be assigned to the local administrator account on the VM deployed.

```
az keyvault secret set --name "ServerAutomationDemo-AppPw" --value $sp.password --vault-na
me "ServerAutomationDemo-KV"
az keyvault secret set --name StandardVmAdminPassword --value "I like azure." --vault-name
"ServerAutomationDemo-KV"
```

*Just so you know, you're using previously defined PowerShell variables in this example. You're providing the service principal password ($sp.password) obtained earlier.*

## Allowing the Pipeline to Access the Key Vault

Next up, the pipeline needs permission to access the key vault. Relax the key vault access policy a bit. Provide the service principal created with *get* and *list* permissions to manage key vault secrets.

```
az keyvault set-policy --name "ServerAutomationDemo-KV" --spn $spIdUri --secret-permission
s get list
```

# Preparing Azure DevOps

You now have all of the Azure resource preparation done. It's time to do some preparation work in Azure DevOps.

## Installing the Pester Extension

The first task to perform is installing the *PesterRunner* Azure DevOps extension. The pipeline will run two sets of Pester tests to ensure the VM ARM deployment was successful. One of the easiest ways to run Pester tests is with the *PesterRunner* extension.

Install the extension using the command below.

```
az devops extension install --extension-id PesterRunner --publisher-id Pester
```

## Creating the Azure DevOps Project

It's now time to create the project in which the pipeline will be created. Creating an Azure DevOps pipeline is a cinch with the Azure CLI. Run the commands below to create the project and set the project as your default.

```
az devops project create --name "ServerAutomationDemo"
az devops configure --defaults project=ServerAutomationDemo
```

## Creating the Service Connections

Your pipeline needs to authenticate to two services - ARM and your GitHub repo. To do this, two service connections must be created.

First, create the ARM service endpoint. The command below will prompt for the service principal password. Be sure first to display it on the console and copy it to your clipboard.

Be sure to fill in your subscription ID, tenant ID and replace the subscription name below.

```
## Run $sp.password and copy it to the clipboard
$sp.Password
```

```
## create the service endpoint
az devops service-endpoint azurerm create --azure-rm-service-principal-id $sp.appId --azur
e-rm-subscription-id "YOURSUBSCRIPTIONIDHERE" --azure-rm-subscription-name 'Adam the Autom
ator' --azure-rm-tenant-id $tenantId --name 'ARM'
```

Next, create a service connection for GitHub. Since the pipeline will be triggered via a Git commit, it will need to be able to read the repo.

At this point is where that GitHub personal access token comes in handy. Below you'll also need to paste in the service principal password again. You're using the same service principal for both service connections.

```
$gitHubServiceEndpoint = az devops service-endpoint github create --github-url 'https://gi
thub.com/adbertram/ServerAutomationDemo' --name 'GitHub' | ConvertFrom-Json

## paste in the GitHub token when prompted
```

## Creating the Variable Group

The pipeline will reference key vault secrets for two passwords. To securely do this, you must create a variable group and link it to the key vault.

First, create the variable group as shown below.

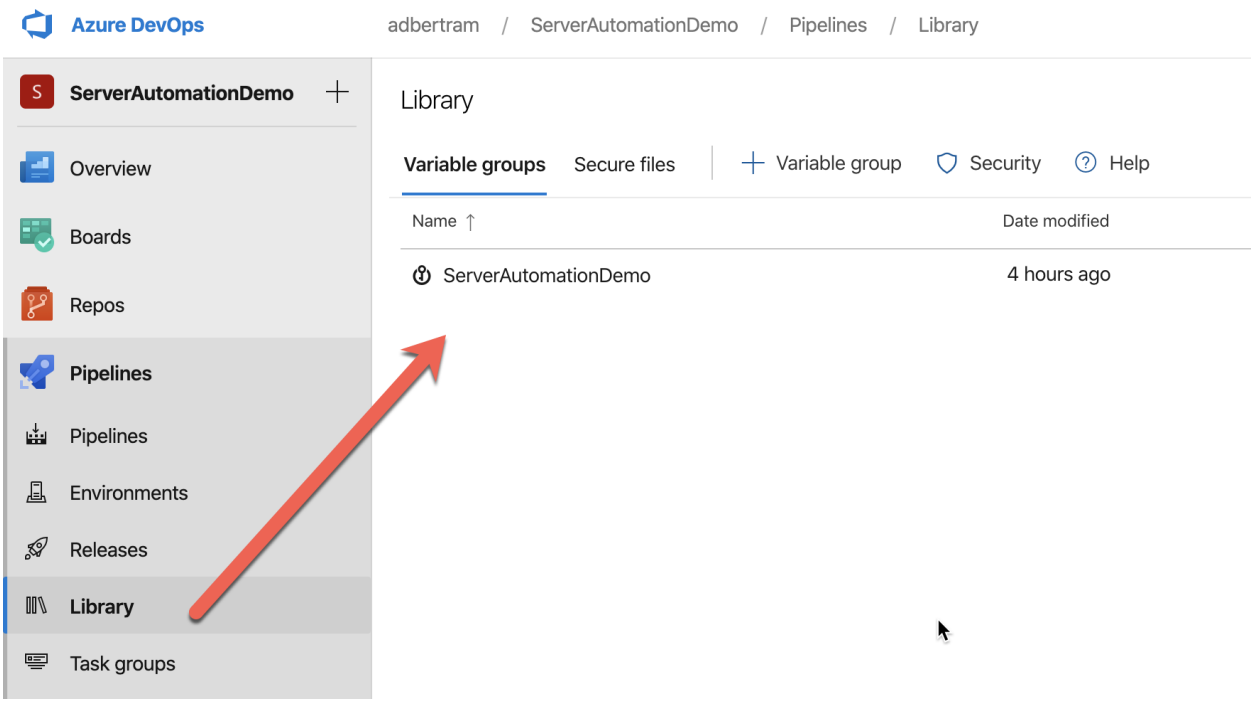
```
az pipelines variable-group create --name "ServerAutomationDemo" --authorize true --variab
les foo=bar
```

*Notice the `foo=bar` variable? This isn't used, but a single variable is required to create the variable group.*

## Linking the Variable Group to a Key Vault

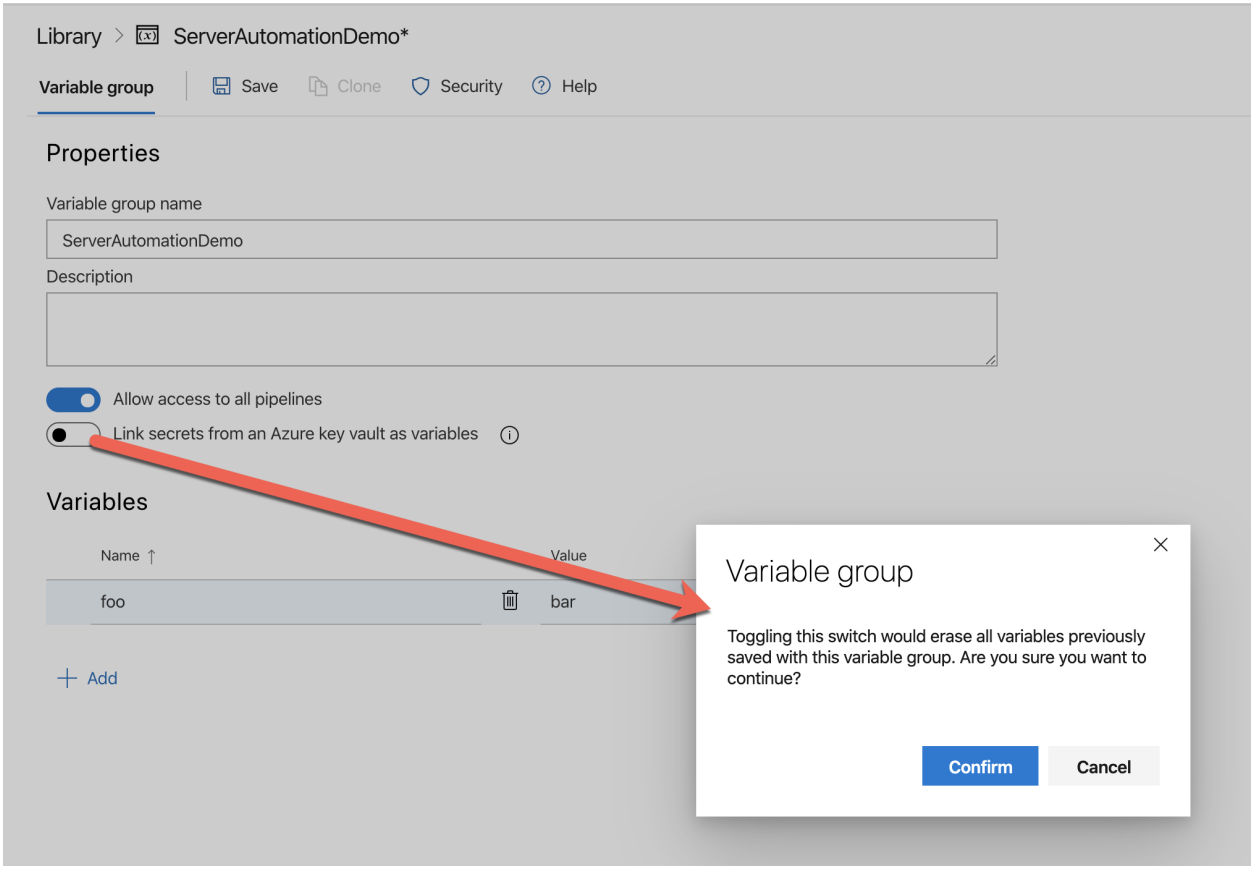
Unfortunately, you will have to go to the Azure DevOps portal. As of the time of this writing, the Azure CLI has no way of linking a key vault to a variable group.

Navigate to the Azure DevOps project and click on **Library**. You should then see the *ServerAutomationDemo* variable group as shown below. Click on the **ServerAutomationDemo** variable group.



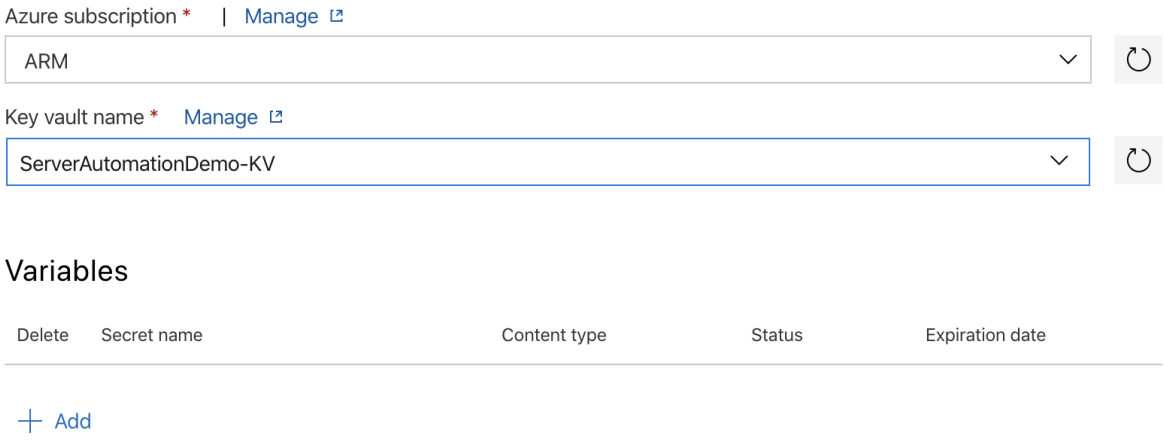

Available variable group

Once in the variable group, click on **Link secrets from an Azure key vault as variables**. Once you do, you'll be warned you'll erase all variables and click **Confirm**. You'll see how to do this below. This action is fine since the *foo* variable was temporary all along anyway.



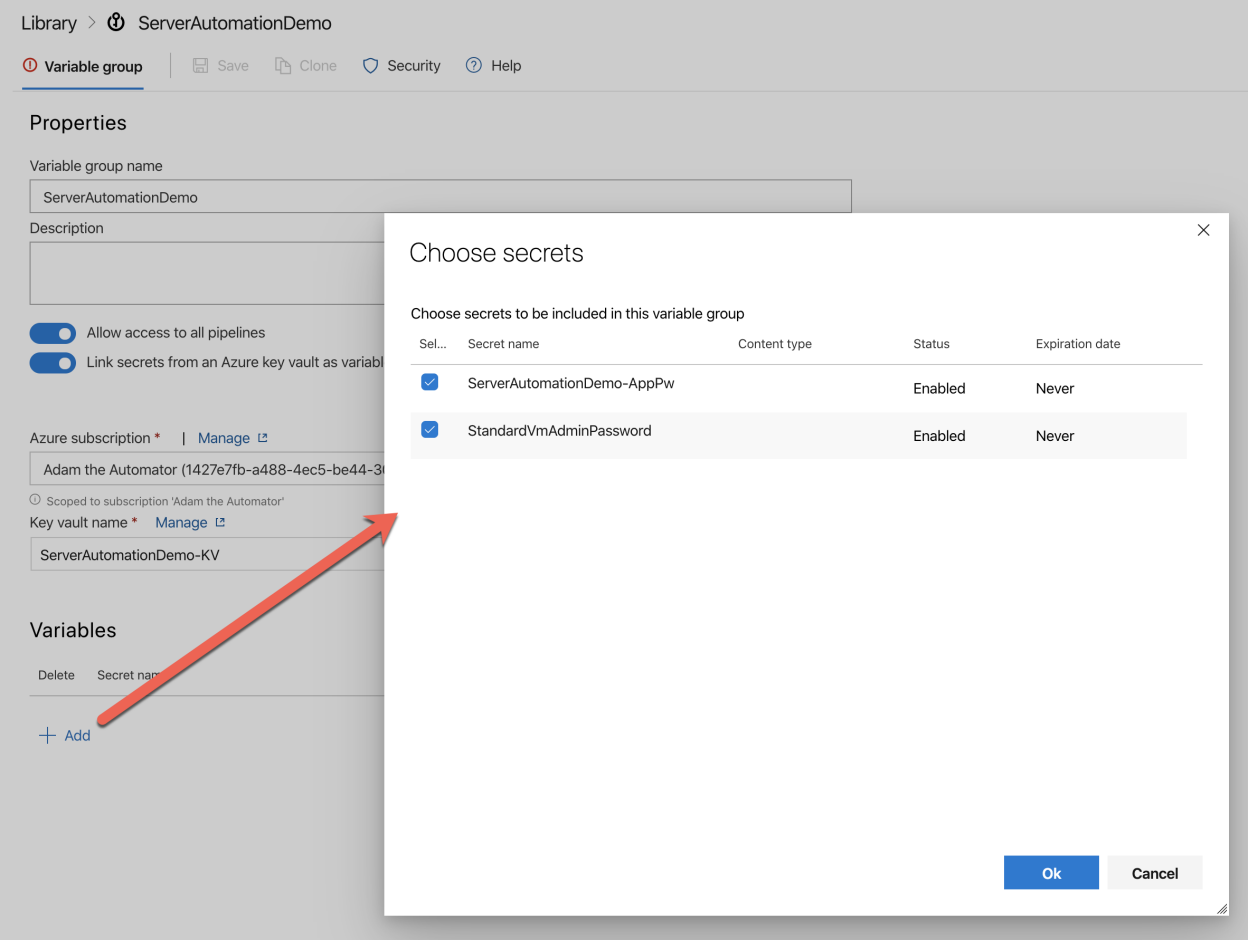

Linking variable group to pipeline

Once confirmed, select the **ARM** service connection and the **ServerAutomationDemo-KV** key vault created earlier, as shown below. Click **Add**.




Setting the service connection

Now check both of the secrets created earlier, as shown below, and click **OK** and **Save** to save the changes.



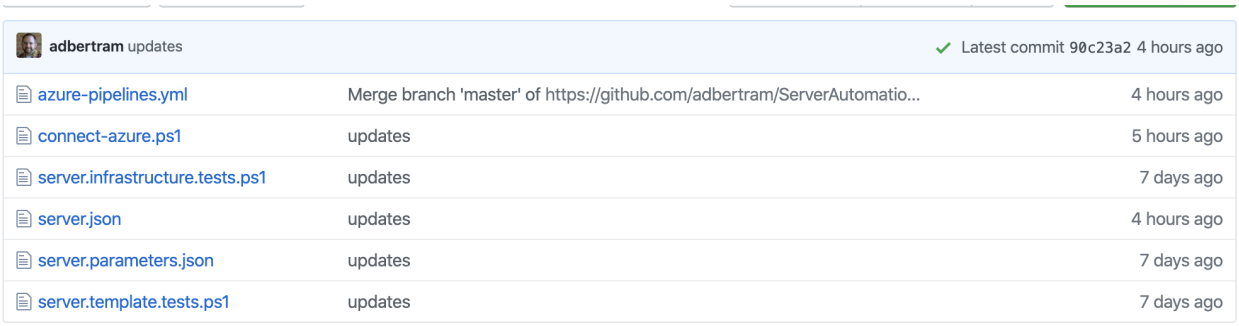

Selecting keyvault secrets for the pipeline to use

# Project Files Overview

If you've made it this far, congratulations! You're now ready to begin building the pipeline. But wait...there's more!

To make building an Azure Pipeline the real world, this Project builds a pipeline complete with "unit" and "acceptance" tests. This makes the tutorial more exciting but also warrants some additional explanation of what's going on.

You'll find a few files in the GitHub repo for this Project, as shown below. Now would be an excellent time to either clone this repo or build your own from the files.




GitHub files list

- *azure-pipelines.yml* - The final YAML pipeline
- *connect-azure.ps1* - PowerShell script to authenticate to an Azure subscription
- *server.infrastructure.tests.ps1* - A simple Pester test to confirm VM configuration is good
- *server.json* - An Azure ARM template to provision a VM
- *server.parameters.json* - An Azure ARM parameters template that provides the ARM template with parameter values.

> 💡 Remember *to replace your subscription ID and key vault name for the key vault IT in the server.parameters.json file.*

- *server.templates.tests.ps1* - Pester "unit" tests to confirm the ARM template is valid

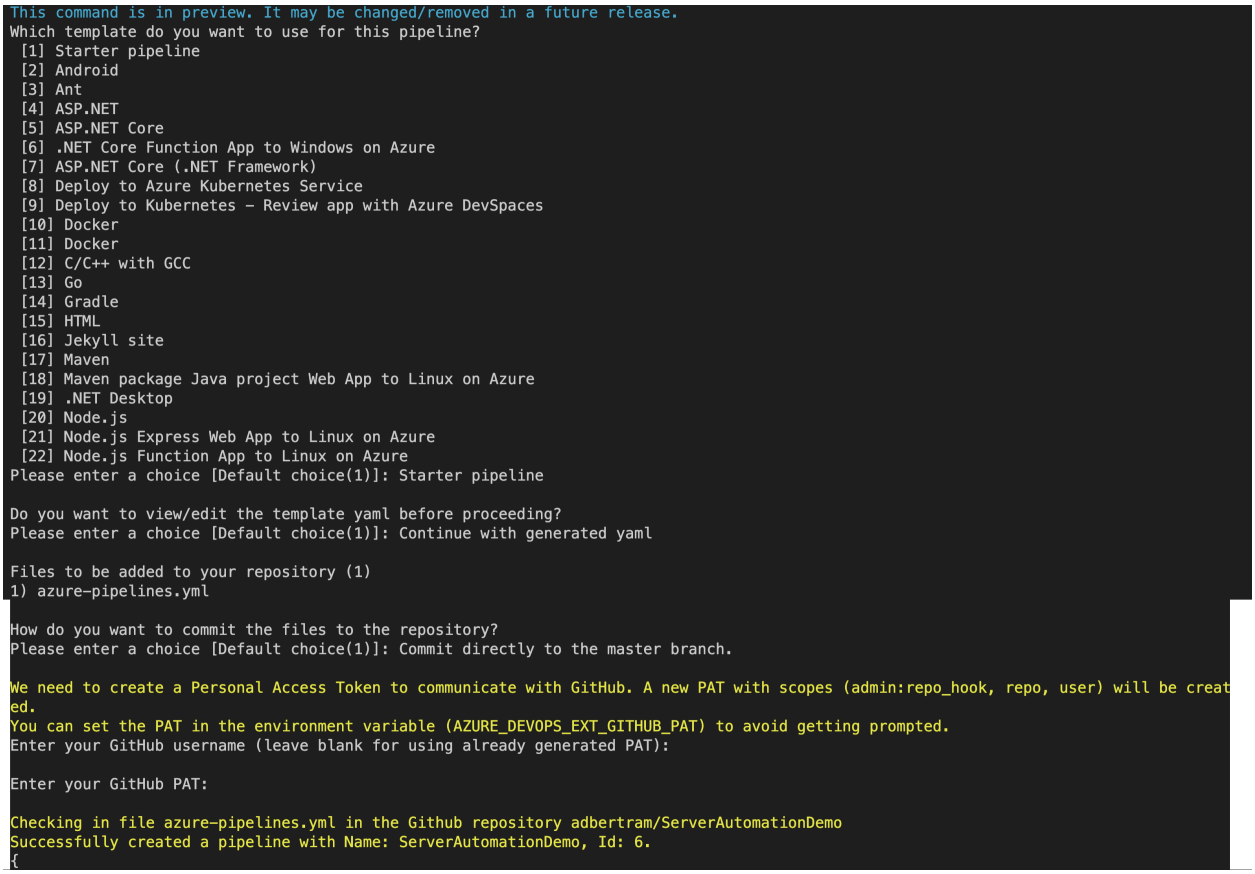
You'll see how each of these files fits together in the pipeline in a little bit.

## Creating the Pipeline

Assuming you've cloned my GitHub repo or set one up on your own, it's time to create the pipeline! To do so, run the `az pipelines create` command. The below command creates a pipeline called *ServerAutomationDemo* using the provided GitHub repo as a trigger. It will look at the *master* branch and use the service connection built earlier.

```
az pipelines create --name "ServerAutomationDemo" --repository "https://github.com/adbertr
am/ServerAutomationDemo" --branch master --service-connection $gitHubServiceEndpoint.id --
skip-run
```

Depending on if you have the *azure-pipelines.yml* file in your GitHub repo, you may or may not receive feedback like below. Either way, your console will look similar. Be sure to have your GitHub personal access token ready!




Creating the Azure DevOps pipeline with the Azure CLI

# YAML Pipeline Review

Your pipeline is ready to run, but it's essential first to understand the YAML pipeline. Take a look at the *azure-pipelines.yml* file. This file is the pipeline when using the multi-stage YAML pipeline feature.

Let's break down the various components that make up this YAML pipeline.

## The Trigger

Since you're building a CI pipeline that automatically runs, you need a trigger. The trigger below instructs the pipeline to run when a commit is detected in the Git master branch.

Notice also the `paths` section. By default, if you don't specifically include or exclude files or directories in a CI build, the pipeline will run when a commit is done on *any* file. Because this project is all built around an ARM template, you don't want to run the pipeline if, for example, you made a tweak to a Pester test.

```
trigger:
  branches:
    include:
      - master
  paths:
    include:
      - server.json
      - server.parameters.json
```

## The Pool

Every build needs an agent. Every build agent needs to run on a VM. In this case, the VM is using the `ubuntu-latest` VM image. This image is the default image defined when the build was originally created. It hasn't been changed due to the "simplicity" of this pipeline.

```
pool:
  vmImage: "ubuntu-latest"
```

### Variables

Next up, we have all of the variables and the variable group. The various tasks in this pipeline require reading values like the Azure subscription ID, tenant ID and the application ID for the service principal, and so on. Rather than replicating static values in each task, they are defined as variables.

Also notice the `group` element. This element is referencing the variable group you created earlier. Please be sure to replace the `subscription_id` and `tenant_id` right now.

Remember in the Creating the Azure Service Principal section you were reminded to save the value of `$sp.appId` somewhere? This is where you'll need it. Assign the value of that service principal application ID to `application_id` as shown below.

```
variables:
    - group: ServerAutomationDemo
    - name: azure_resource_group_name
      value: "ServerProvisionTesting-$(Build.BuildId)"
    - name: subscription_id
      value: "XXXXXXXXXXXXX"
    - name: application_id
      value: "XXXXXXXXXXXXX"
    - name: tenant_id
      value: "XXXXXXXXXXXX"
```

*Note the value of the `azure_resource_group_name` variable. Inside of that value, you'll see `$(Build.BuildId)`. This is a system variable that represents the build ID of the current job. In this context, it is being used to ensure the temporary resource group created is unique.*

## PowerShell Prep Tasks

The next set of tasks invokes PowerShell code. This pipeline example uses PowerShell to create and remove a temporary resource group for testing purposes. You'll see two examples of invoking PowerShell code in these deployment tasks.

The first task invokes a script called *connect-azure.ps1* in the GitHub repo. This task authenticates to the Azure subscription for the subsequent Azure PowerShell commands.

This Azure PowerShell connect task calls the script and passes a key vault secret value (`ServerAutomationDemo-AppPw`) and the pipeline variables `subscription_id`, `application_id`, and `tenant_id`.

The second task is running PowerShell code *inline,* meaning a script doesn't exist. Instead, PowerShell code is defined in the YAML pipeline using the value of the `azure_resource_group_name` pipeline variable.

```
- task: PowerShell@2
  inputs:
    filePath: "connect-azure.ps1"
    arguments: '-ServicePrincipalPassword "$(ServerAutomationDemo-AppPw)" -SubscriptionId
 $(subscription_id) -ApplicationId $(application_id) -TenantId $(tenant_id)'
- task: PowerShell@2
  inputs:
```

```
    targetType: "inline"
    script: New-AzResourceGroup -Name $(azure_resource_group_name) -Location eastus -Force
```

## Pester Template Test

Next up, we have the first Pester test. In a CI/CD pipeline such as this, it's important to have a few different layers of tests. If you were creating a pipeline for a software project, you may create various unit tests.

Since this example pipeline is built around a single ARM VM deployment, the first "unit" tests will be to test the validity of the JSON template. Inside of the *server.templates.tests.ps1* file is where you can add as many tests on the ARM template file.

Please take a look below, the pipeline is using various system variables. These variables reference the file location of the files once they get onto the build agent.

The PesterRunner task is sending the test results out to an XML file which will then be read later in the pipeline.

```
- task: Pester@0
  inputs:
    scriptFolder: "@{Path='$(System.DefaultWorkingDirectory)/server.template.tests.ps1'; P
arameters=@{ResourceGroupName='$(azure_resource_group_name)'}}"
    resultsFile: "$(System.DefaultWorkingDirectory)/server.template.tests.XML"
    usePSCore: true
    run32Bit: False
```

## The ARM VM Deployment

We have now come to the ARM deployment. Since the entire pipeline is built around the ability to deploy a VM, this one is important! This task deploying the ARM template provides all of the required attributes to make it happen.

Note the `deploymentOutputs: arm_output` attribute. In the next step, a task needs to connect to the VM that was deployed. A great way to get this VM's DNS name or IP address is by returning it via the ARM deployment. The `deploymentOutputs` option creates a pipeline variable that can be referenced in other tasks.

```
- task: AzureResourceManagerTemplateDeployment@3
  inputs:
```

```
    deploymentScope: "Resource Group"
    azureResourceManagerConnection: "ARM"
    subscriptionId: "1427e7fb-a488-4ec5-be44-30ac10ca2e95"
    action: "Create Or Update Resource Group"
    resourceGroupName: $(azure_resource_group_name)
    location: "East US"
    templateLocation: "Linked artifact"
    csmFile: "server.json"
    csmParametersFile: "server.parameters.json"
    deploymentMode: "Incremental"
    deploymentOutputs: "arm_output"
```

## Pester "Acceptance" Test

Once the VM has been deployed, you must ensure it was correctly deployed with an "integration" or "acceptance" test. This PesterRunner task is invoking Pester and running another set of infrastructure-related tests to ensure the VM was deployed successfully.

Notice that we're passing in the value of the output of the ARM deployment via the `ArmDeploymentJsonOutput` parameter. The Pester test script file has a parameter defined that takes the value and reads the DNS hostname of the VM.

```
 - task: Pester@0
    inputs:
      scriptFolder: "@{Path='$(System.DefaultWorkingDirectory)/server.infrastructure.test
s.ps1'; Parameters=@{ArmDeploymentJsonOutput='$(arm_output)'}}"
      resultsFile: "$(System.DefaultWorkingDirectory)/server.infrastructure.tests.XML"
      usePSCore: true
      run32Bit: False
```

You can see below what the *server.infrastructure.tests.ps1* PowerShell script looks like. Notice that it reads the VM's DNS hostname to run a simple open port check.

```
$ArmDeploymentOutput = $ArmDeploymentJsonOutput | convertfrom-json

## Run the tests
describe 'Network Connnectivity' {
    it 'the VM has RDP/3389 open' {
Test-Connection -TCPPort 3389 -TargetName $ArmDeploymentOutput.hostname.value -Quiet | sho
uld -Be $true
    }
}
```

## "Acceptance" Test Cleanup

The only reason the pipeline deployed any infrastructure was to test the validity of the ARM template. Because this infrastructure is only temporary, it needs to be cleaned up. In the last PowerShell task, the pipeline removes the existing resource group and everything in it.

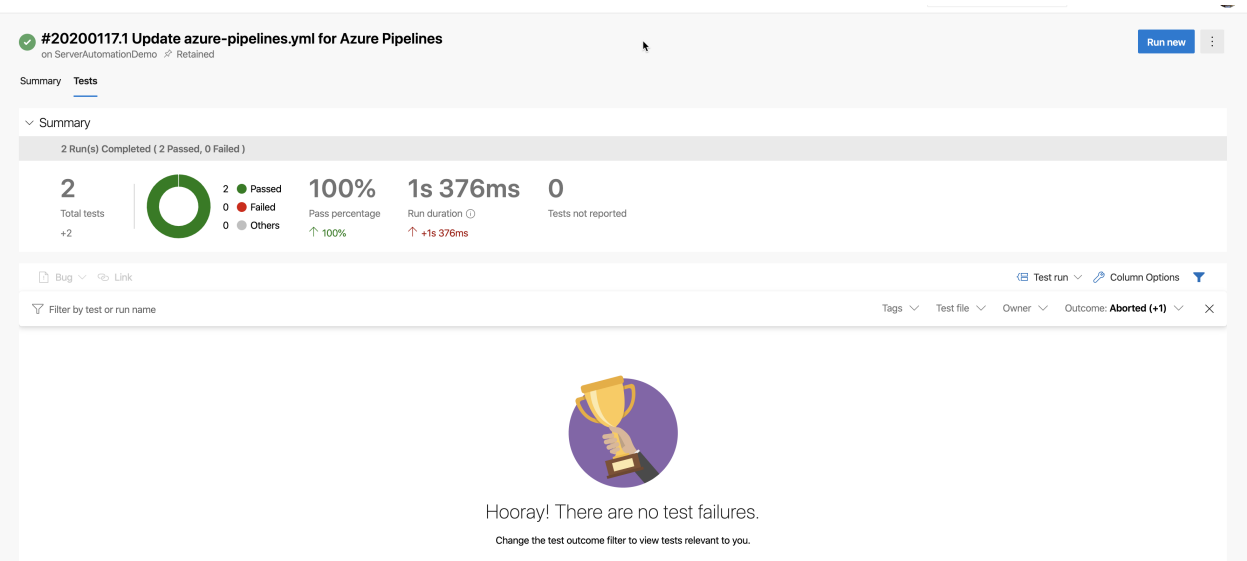
```
- task: PowerShell@2
  inputs:
    targetType: "inline"
    script: Get-AzResourceGroup -Name $(azure_resource_group_name) | Remove-AzResourceGrou
p -Force
```

### Pester Test Publishing

And finally, we have come to the last set of tasks. Azure Pipelines has a task called *Publish Test Results*. This task reads an XML file on the build agent and displays test results in Azure DevOps. This is a handy way to quickly see the results of all tests run.

```
- task: PublishTestResults@2
  inputs:
    testResultsFormat: "NUnit"
    testResultsFiles: "$(System.DefaultWorkingDirectory)/server.infrastructure.tests.XML"
    failTaskOnFailedTests: true

- task: PublishTestResults@2
  inputs:
    testResultsFormat: "NUnit"
    testResultsFiles: "$(System.DefaultWorkingDirectory)/server.template.tests.XML"
    failTaskOnFailedTests: true
```

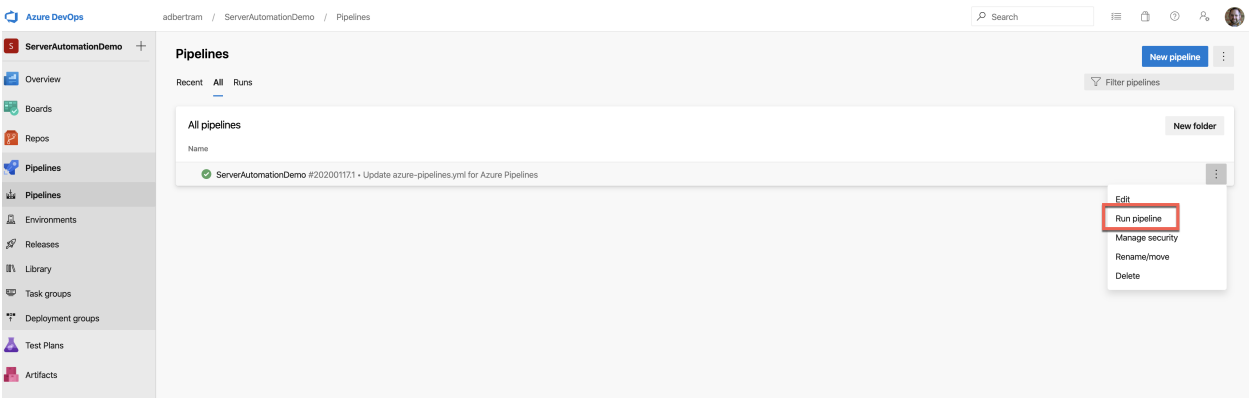



The **Tests** section of a pipeline run

# Using the Azure DevOps Pipeline

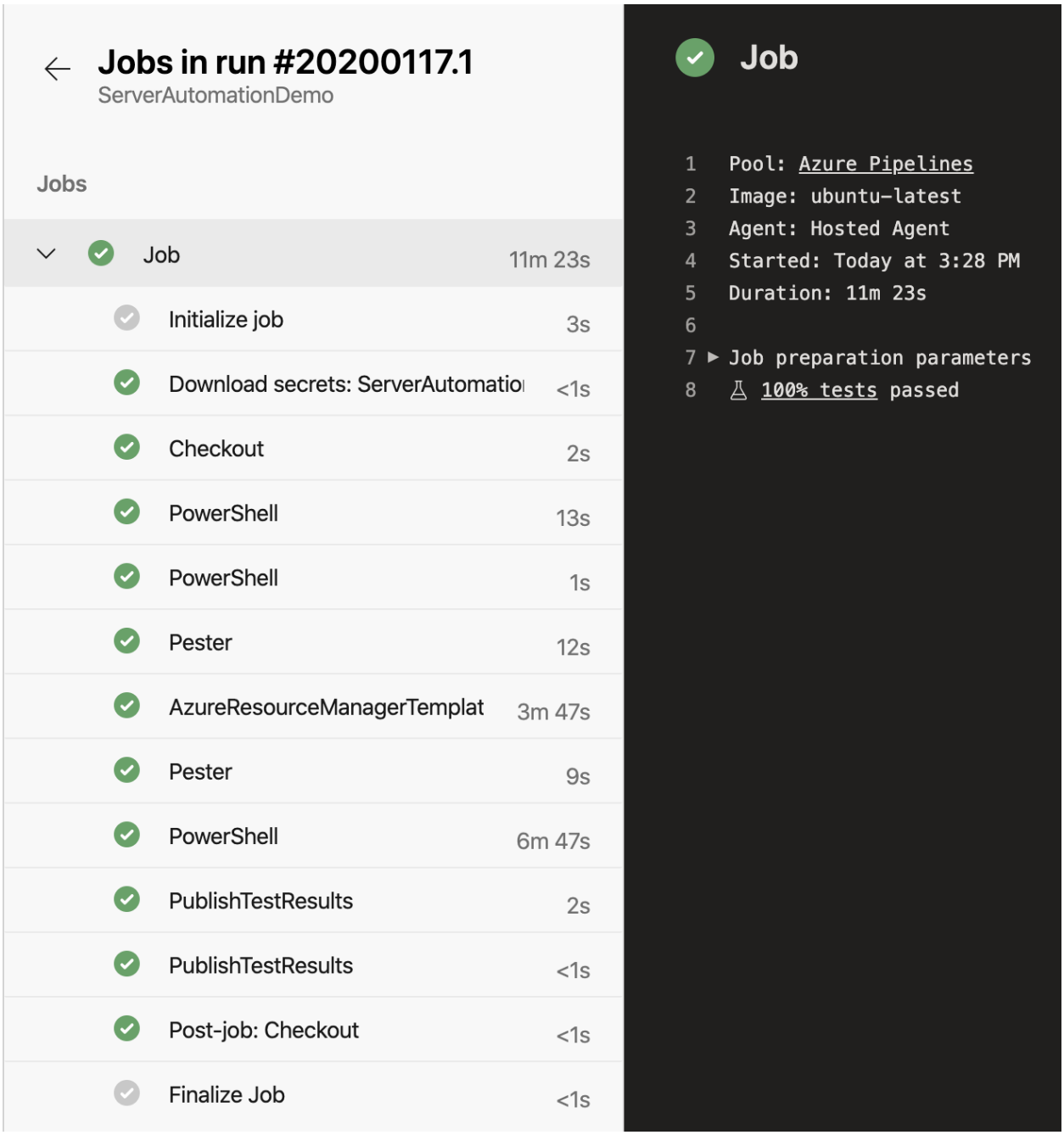
*Finally*, we're ready to run the pipeline and see how it works. In the Azure DevOps web UI, please make sure you're in the *ServerAutomationDemo* project. Once here, click on **Pipelines,** and then you should see the *ServerAutomationDemo* pipeline.

One way to run the pipeline is to click on the three dots on the far right, as shown below. Then, click on **Run pipeline**. This will kick off the automation goodness.




Running a pipeline

The pipeline will chug along and run each task as instructed. By the end, you should see all green check marks for each task performed by the job, as shown below.




Successful job task execution

## Cleaning Up

Once you've fiddled around with the pipeline and everything you've accomplished here, you should clean things up. After all, this was just meant to be a tutorial and *not* a production task!

Below are some commands to clean up everything built in this Project. This code removes the service principal, Azure AD application, the resource group and everything in it, and the Azure DevOps project.

```
$spId = ((az ad sp list --all | ConvertFrom-Json) | ? { '<https://ServerAutomationDemo>' -
in $_.serviceprincipalnames }).objectId
az ad sp delete --id $spId

## Remove the resource group
az group delete --name "ServerAutomationDemo" --yes --no-wait

## remove project
$projectId = ((az devops project list | convertfrom-json).value | where { $_.name -eq 'Ser
verAutomationDemo' }).id
az devops project delete --id $projectId --yes
```

## Summary

This Project was meant to give you a peek into building an actual Azure DevOps infrastructure automation pipeline. Even though there are countless other ways to build pipelines such as this, the skills you've learned in this Project should assist you through many different configurations.

Now get out there and start doing more automating!