Using Ansible with GitHub Actions and Azure DevOps

In this chapter, we will start running Ansible in the cloud instead of our local machines, something we have been doing up to this point.

First, this chapter will examine two services I often use during my day job:

* Running GitHub Actions
* Running pipelines in Azure DevOps

Before we move on, we will examine tools designed to execute Ansible from a central location in [*Chapter 16*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/16), *Introducing Ansible AWX and Red Hat Ansible Automation Platform*.

Neither of the two services we will be looking at has what you would call native support for Ansible; however, they both provide ephemeral compute resources that can be configured using YAML, which you can ship alongside your playbook code.

This chapter will cover a more complex playbook in both **GitHub Actions** and **Azure DevOps**. We will also discuss some considerations when running Ansible away from your machine.

So, rather than discussing it anymore, let’s dive straight in and look at GitHub Actions.

Technical requirements

If you are following along with the example code that we will be working through, then you will need access to a GitHub and an Azure DevOps account, as well as an Azure account since we will be launching a WordPress instance running in Azure as part of this chapter.

You can find a complete copy of the playbook, GitHub Action config, and Azure DevOps Pipeline code accompanying this chapter in this book’s GitHub repository at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter15.

GitHub Actions

GitHub Actions is a comprehensive platform for **continuous integration** (**CI**) and**continuous delivery** (**CD**) from GitHub. It enables you to automate your build, test, and deployment pipeline while hosting your code and GitHub’s exhaustive suite of code management tools. Using GitHub Actions, you can define custom workflows that automatically build and test every pull request made to your repository or deploy merged pull requests to production.

GitHub Actions offers more than just DevOps functionality as it is closely integrated with GitHub. This allows you to run workflows in response to other repository events. For instance, you can have a workflow that adds relevant labels automatically when a new issue is created in your repository.

With GitHub Actions, you’re in control. You can run your workflows using GitHub’s Linux, Windows, and macOS virtual machines. You can also take full charge and operate self-hosted runners in your own data center or cloud infrastructure.

We will create a GitHub Action workflow to utilize the Linux agents hosted by GitHub.

Preparation

We need to configure a few things before we can start working through our GitHub Action workflow code:

1. Create a GitHub repository to host our code and workflow.
2. Generate an SSH key pair; this will be used to access our Azure-hosted virtual machine instance from the GitHub-hosted compute resource when the workflow runs.
3. Configure some repository secrets that will be used in our workflow; these will store things such as our Azure credentials and the SSH key pair we created.
4. Copy the files from https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter15 to your new repo and run the workflow.

Let’s look at these steps in more detail.

Creating a repository

Let’s start by creating a repository in GitHub that we will use to host our code and workflow.

First, you need to log into GitHub. Once you’ve logged in, go to **Repositories** and then click on the **New** button; you will be taken to the **Create a new repository** page, where you need to update the following:

* **Owner**: Here, you need to select an owner for the repository. This will typically be your GitHub user; however, if you belong to any organization, you may have the option of creating the repository under one of those organizations. If you do that, please ensure that you have permission to do so as we will be spinning up temporary compute resources, which may not be allowed by your organization’s admins.
* **Repository name**: I recommend using something descriptive, such as *Learn-Ansible-Second-Edition-Chapter15*.
* **Description**: While this is optional, it is always best to add one; for example, let’s add *Following along with*[*Chapter 15*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/15)*of Learn Ansible*.
* **Public** or **Private**: I recommend setting your repository’s visibility to *private*.

You can leave the remaining options as-is and then click on the **Create repository** button at the end of the form. Once the repository has been created, you should be presented with a page that looks like this:

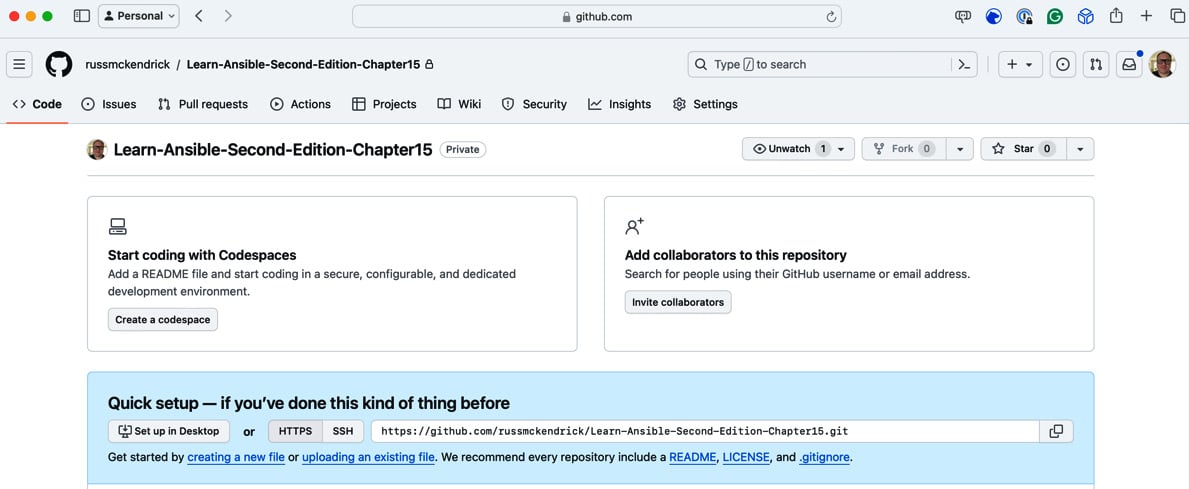


Figure 15.1 – Our new repository

Let’s move on to the next step.

Generating the SSH key pair and Azure Service Principle

We need to generate an SSH key pair and an Azure Service Principle before adding secrets to our newly created repository.

Information

Remember to run the commands in Windows Subsystem for Linux if you’re following along on a Windows machine.

To do this, open a Terminal and run the following command:

$ ssh-keygen -t rsa -C "learnansible" -f ./id\_rsa

CopyExplain

When prompted to enter a passphrase, just hit *Enter*; we don’t want to use one. This should give you two files: one called id\_rsa, which contains the private portion of our key – please keep this private – and another called id\_rsa.pub. As its name implies, it includes the public portion of our SSH key.

Next, we need to generate an Azure Service Principle and grant permissions to our Azure subscription.

From [*Chapter 7*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/7), *Ansible Windows Modules*, and [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*, we used the Azure command-line tool to log in using our Azure credentials. However, when interacting with Azure using services such as GitHub Actions, we don’t want to use our credentials as they will be locked down with multi-factor authentication, and you don’t want to hand out your credentials.

To get around this, we can create a service principal and grant it permissions to the Azure subscription so that it can launch resources from the GitHub Action.

To create the service principle, you need to log into Azure using the Azure CLI by running the following command:

$ az login

CopyExplain

If you are already logged in, run the following:

$ az account list

CopyExplain

Both commands will return a list of subscription IDs your account can access. Please make a note of the ID; we will need it momentarily.

Here’s an example of the sort of output you can expect to see; this is the JSON that is being returned by the API request that the Azure CLI has made:

{

  "environmentName": "AzureCloud",

  "id": "e80d5ad9-e2c5-4ade-a866-bcfbae2b8aea",

  "isDefault": true,

  "name": "My Subscription",

  "state": "Enabled",

  "tenantId": "c5df827f-a940-4d7c-b313-426cb3c6b1fe",

  "user": {

    "name": "account@russ.foo",

    "type": "user"

  }

}

CopyExplain

The information we are after is labeled as id against the subscription to which we would like to grant the service principal access. Using the preceding example, the command I would need to run would be as follows:

$ az ad sp create-for-rbac –name sp-learn-ansible –role contributor –scopes /subscriptions/e80d5ad9-e2c5-4ade-a866-bcfbae2b8aea

CopyExplain

When you run this command, replace the subscription ID in the scope with your own.

The output you get will look something like this; please note it down as you will not be able to retrieve the password again:

Creating 'contributor' role assignment under scope '/subscriptions/e80d5ad9-e2c5-4ade-a866-bcfbae2b8aea'

The output includes credentials that you must protect. Be sure that you do not include these credentials in your code or check the credentials into your source control. For more information, see https://aka.ms/azadsp-cli

{

  "appId": "2616e3df-826d-4d9b-9152-3de141465a69",

  "displayName": "sp-learn-ansible",

  "password": "Y4j8Q~gVO\*NoTaREalPa55w0rdpP-pdaw",

  "tenant": "c5df827f-a940-4d7c-b313-426cb3c6b1fe"

}

CopyExplain

Also, as I am sure you will have already guessed, none of the information in the preceding examples is valid data, so please use your values in the next section.

GitHub personal access token

There is one more set of credentials we need to generate; because our GitHub repository is set to private, we need to be able to authenticate to check the code out and write logs back to the repository during the workflow run. To do this, we will need to generate a personal access token.

A personal access token for GitHub is a secure, revocable, and customizable credential that allows you to authenticate with GitHub and access its API or command-line tools without using your main account password.

Rather than documenting the process here, as GitHub is moving from classic to fine-grained tokens at the time of writing, an up-to-date copy of the documentation can be found at https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens.

For our purposes, you need to name your token, select just your repository, and grant it the following access:

* **Contents**: Read-only
* **Metadata**: Read-only; this will be selected automatically once the permission is set

Once you have your token, please note it somewhere secure; it will not be displayed again.

Adding secrets to the repository

Go back to the repository in GitHub by choosing **Settings** | **Secrets and Variables** | **Actions**. Click the **New repository secret** button for each of the secrets listed in the following table. Please make sure that you call each secret as per the following naming conventions since our workflow code references these secrets by their name:

|  |  |
| --- | --- |
| **Secret Name** | **Secret Content** |
| ARM\_CLIENT\_ID | This is the appId value from when you create the service principle. In this example, this would be 2616e3df-826d-4d9b-9152-3de141465a69. |
| ARM\_CLIENT\_SECRET | This is the password value that was given when you created the service principle. In this example, this would be Y4j8Q~gVO\*NoTaREalPa55w0rdpP-pdaw. |
| ARM\_SUBSCRIPTION\_ID | This is your Azure subscription ID; use the one you granted the service principal access to. In this example, this would be e80d5ad9-e2c5-4ade-a866-bcfbae2b8aea. |
| ARM\_TENANT\_ID | This is the ID of the tenant value listed when you created the service principle. In this example, this would be c5df827f-a940-4d7c-b313-426cb3c6b1fe. |
| SSH\_PRIVATE\_KEY | Open the id\_rsa file in a text editor and copy and paste the contents here. |
| SSH\_PUBLIC\_KEY | Open the id\_rsa.pub file in a text editor and copy and paste the contents here. |
| GH\_PAT | This should contain your GitHub personal access token. |

Table 15.1 – Information needed for GitHub Actions

Once they have all been added, your **Actions secrets and variables** page should look something like this:

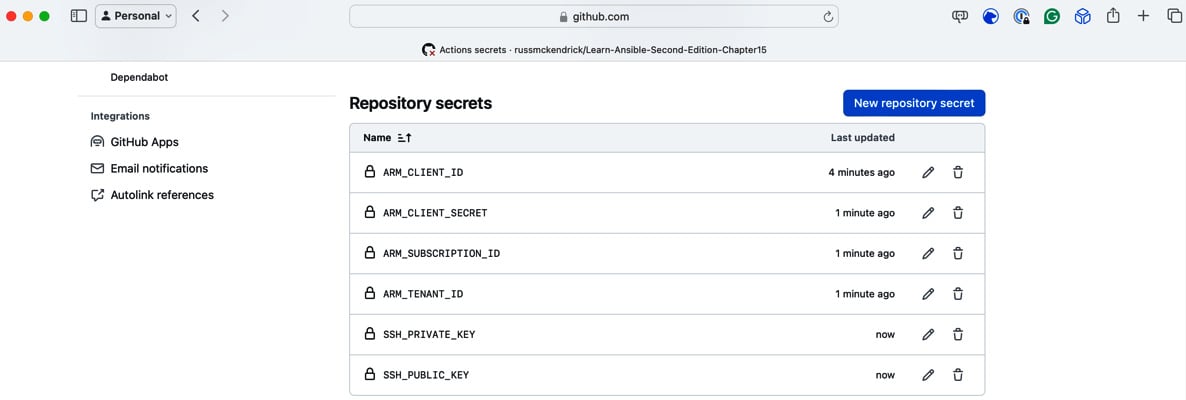


Figure 15.2 – All of the repository secrets have been added

Now that we have all the basic configurations for the GitHub Action, let’s look at the workflow itself.

Understanding the GitHub Action workflow

The workflow file, which lives in the .github/workflows/action.yml file, contains, as its name suggests, the YAML code containing the jobs, steps, and tasks that will be executed during the workflow run. In our case, the workflow will execute the following two jobs, with each job being made up of multiple steps:

* Scan the Ansible Playbook:
  1. Check out the code.
  2. Create a folder to store the scan results.
  3. Run a KICS scan on the checked-out code.
  4. Upload a copy of the results to GitHub.

Now, if KICS detects a problem with our playbook, it will report an error, and the workflow will stop here – if everything looks good with the KICS scan, then the workflow will proceed by running the following job:

* Install and run the Ansible Playbook:

1. Check if a cached version of our Ansible modules and Python packages is available.
2. If not cached, download and install the Ansible Azure modules and the supporting Python packages.
3. Check out the code.
4. Log into Azure using the Azure CLI and the service principle we created.
5. Set the SSH key.
6. Run the Ansible Playbook, logging the output of the Playbook so that we can store a copy alongside the scan results in the workflow logs.
7. Upload the Playbook execution summary.

Now that we know what the workflow will do, let’s dive into the code. We’ll start with some basic configuration:

1. The first line disables a KICS check – while the workflow does not form part of our Playbook, it is stored in the repository and will be scanned as part of the workflow’s execution:
2. # kics-scan disable=555ab8f9-2001-455e-a077-f2d0f41e2fb9
3. name: "Ansible Playbook Run"
4. env:
5. FAIL\_ON: "medium"

  RESULTS\_DIR: "results-dir"

CopyExplain

We are also setting the name of the workflow, which is how it will appear in the GitHub web interface, before finally setting up some variables that we will use during the workflow’s execution.

1. Next up, we have the configuration that defines the workflow that should run; for our needs, we will run the workflow each time the code is committed to the main branch:
2. On:
3. push:
4. branches:

      - main

CopyExplain

1. Next up, we must define our first job, which is the one that scans the Playbook code:
2. jobs:
3. scan\_ansible\_playbook:
4. name: "Scan Ansible Playbook"
5. runs-on: ubuntu-latest
6. defaults:
7. run:

        shell: bash

CopyExplain

As you can see, we are defining it as scan\_ansible\_playbook, which runs on the latest version of the Ubuntu image supplied by GitHub, and the default action for tasks is to run bash. With the job defined, we can move on to the next steps.

1. We start with the ones that check out the code and create the directory where we are going to be storing the results of the scan we will be running:
2. steps:
3. - name: "Checkout the code"
4. uses: "actions/checkout@v4"
5. with:

          token: "${{secrets.GH\_PAT}}"

CopyExplain

1. The step downloads a copy of the repository in which the workflow is hosted; as you can see, we are using ${{secrets.GH\_PAT}}. We will look at secret variables a little later. Now, we must create the folder:
2. - name: "Create the folder for storing the scan results"

        run: mkdir -p ${{env.RESULTS\_DIR}}

CopyExplain

The section step creates a directory whose name is referenced as the RESULTS\_DIR environment variable, which we defined in the top section of the workflow file.

1. When referencing an environment variable, we use the ${{env.VARIABLE\_NAME}} format. So, in our case, we are using ${{env.RESULTS\_DIR}}. In the next step, we have a dedicated task for running KICS, which is managed and maintained by Checkmarx:
2. - name: "Run kics Scan"
3. uses: "checkmarx/kics-github-action@v1.7.0"
4. with:
5. path: "./"
6. output\_path: "${{env.RESULTS\_DIR}}"
7. output\_formats: "json,sarif"
8. fail\_on: "${{ env.FAIL\_ON }}"

          enable\_jobs\_summary: true

CopyExplain

As you can see, we are instructing the task to output the JSON and SARIF files, **SARIF**, which stands for **Static Analysis Results Interchange Format**, is a standardized JSON-based file format for the output of static analysis tools that allows you to share and integrate analysis results between different tools and platforms. The results are outputted to the ${{env.RESULTS\_DIR}} directory we created in the previous step and also for the workflow fail if the results of the scan contain anything with a severity defined in ${{ env.FAIL\_ON }}. We set this to medium at the start of the workflow file.

1. Now that we have completed the scan, we can review the workflow code for the job that installs and runs Ansible. This is called run\_ansible\_playbook:
2. run\_ansible\_playbook:
3. name: "Install Ansible and run Playbook"
4. runs-on: ubuntu-latest
5. needs: scan\_ansible\_playbook
6. defaults:
7. run:

        shell: bash

CopyExplain

As you can see, the job is defined the same as the first job, with one exception: we have added a needs line with a value of scan\_ansible\_playbook. This instructs the job to only run once scan\_ansible\_playbook has completed with a successful status.

1. The step of the job checks for the presence of three folders; if they exist, a cached version of those folders will be used, meaning that once the workflow has been run once, subsequent executions will be much quicker as we don’t have to install the Ansible Galaxy modules and their requirements each time the workflow runs:
2. steps:
3. - name: "Cache Ansible collections and Python packages"
4. uses: actions/cache@v4
5. with:
6. path: |
7. ~/.ansible/collections
8. ~/.cache/pip
9. /home/runner/.local/lib/python3.10/site-packages
10. key: ${{ runner.os }}-ansible-collections-and-python-packages
11. restore-keys: |

            ${{ runner.os }}-ansible-collections-and-python-packages

CopyExplain

1. Next up, we have the step that checks out our repo:
2. - name: "Checkout the code"
3. id: "checkout"

        uses: "actions/checkout@v4"

CopyExplain

You might be wondering, “*Why do we need to check out the code again? We already did that during the last job.*” This is a great question.

The answer is that the compute resource that ran the job was terminated when the last job finished running, and all data was lost. When the current job started, a new resource was launched, and we started again with a completely fresh installation.

1. The next step in the workflow uses the Azure/login@2 task to install the Azure CLI if it’s not already installed and then log in using the service principal information we defined as repository secrets earlier in this chapter:
2. - name: "Login to Azure using a service principal"
3. uses: "Azure/login@v2"
4. with:

          creds: '{"clientId":"${{secrets.ARM\_CLIENT\_ID }}","clientSecret":"${{secrets.ARM\_CLIENT\_SECRET }}","subscriptionId":"${{secrets.ARM\_SUBSCRIPTION\_ID }}","tenantId":"${{secrets.ARM\_TENANT\_ID }}"}'

CopyExplain

We need to embed secrets using the ${{ secrets.SECRET\_NAME }} format. Here, we are using the following:

* + ${{secrets.ARM\_CLIENT\_ID }}
  + ${{secrets.ARM\_CLIENT\_SECRET}}
  + ${{ secrets.ARM\_SUBSCRIPTION\_ID }}
  + ${{ secrets.ARM\_TENANT\_ID }}

Because these are all defined as secrets, the values will never appear in any of the Pipeline run logs.

This means that while we know the values, someone else who has permission to run the workflow will never need to be told the credentials for our service principle as they can consume the secrets. They will also never accidentally be exposed to them if they check any logs or try and output them due to the workflow’s execution as they will be automatically redacted.

1. The final step before we run Ansible is to add and configure the SSH key pair to our host:
2. - name: "Setup SSH key for Ansible"
3. id: "add-ssh-key"
4. run: |
5. mkdir ~/.ssh
6. chmod 700 ~/.ssh/
7. echo "${{ secrets.SSH\_PRIVATE\_KEY }}" > ~/.ssh/id\_rsa
8. chmod 600 ~/.ssh/id\_rsa
9. echo "${{ secrets.SSH\_PUBLIC\_KEY }}" > ~/.ssh/id\_rsa.pub
10. chmod 644 ~/.ssh/id\_rsa.pub

          cat  ~/.ssh/id\_rsa.pub

CopyExplain

1. The SSH key pair is the final piece we needed. Now, we can run Ansible:
2. - name: "Run the playbook (with ansible-playbook)"
3. id: "ansible-playbook-run"
4. continue-on-error: true
5. run: |
6. ansible-playbook -i inv site.yml 2>&1 | tee ansible\_output.log
7. echo "summary<<EOF" >> $GITHUB\_OUTPUT
8. echo "## Ansible Playbook Output" >> $GITHUB\_OUTPUT
9. echo "<details><summary>Click to expand</summary>" >> $GITHUB\_OUTPUT
10. echo "" >> $GITHUB\_OUTPUT
11. echo "\`\`\`" >> $GITHUB\_OUTPUT
12. cat ansible\_output.log >> $GITHUB\_OUTPUT
13. echo "\`\`\`" >> $GITHUB\_OUTPUT
14. echo "</details>" >> $GITHUB\_OUTPUT
15. echo "EOF" >> $GITHUB\_OUTPUT
16. env:

          ANSIBLE\_HOST\_KEY\_CHECKING: "False"

CopyExplain

As you can see, there is slightly more to running Ansible here than we have been doing on our local machines. The reason we are running the Ansible playbook is to capture its output and format the output so that it can be displayed in the GitHub Actions job log.

Here’s a breakdown of what’s happening:

* **Name**: This step is named Run the playbook (with ansible-playbook) for clarity in the workflow’s execution log.
* **ID**: The step is given an identifier of ansible-playbook-run so that we can refer to this step’s outputs in the subsequent step.
* **Continue on Error**: By setting continue-on-error to true, we are allowing the workflow to continue even if this step encounters an error. This is useful for ensuring that the workflow can proceed to steps that might, for example, provide diagnostic information or perform cleanup actions, even if the Ansible playbook fails.
* **Run**: This key starts a multi-line script block that’s executed in the jobs shell. The script does the following:

ansible-playbook -i inv site.yml 2>&1 | tee ansible\_output.log

CopyExplain

This command runs the Ansible playbook defined in site.yml with an inventory file, inv. The 2>&1 part redirects stderr to stdout, so both standard output and errors from the ansible-playbook command are piped to the tee command. tee ansible\_output.log writes the output to ansible\_output.log and displays it in the workflow’s log for real-time monitoring.

Subsequent echo commands and cat append a formatted summary of the Ansible output to the special GITHUB\_OUTPUT environment variable. As you may have noticed, we are mostly using Markdown to format the text.

* **Env**: The env section defines environment variables for this step. ANSIBLE\_HOST\_KEY\_CHECKING: "False" disables Ansible’s SSH host key checking. This option is often used in automated environments to avoid manual interventions.

The final step in our workflow takes the output of the previous step and outputs it to $GITHUB\_STEP\_SUMMARY. This is a special variable that’s used by a GitHub Actions workflow to record the results of a step in the workflow executions log:

      - name: "Publish Ansible Playbook run to Task Summary"

        env:

          SUMMARY: ${{ steps.ansible-playbook-run.outputs.summary }}

        run: |

          echo "$SUMMARY" >> $GITHUB\_STEP\_SUMMARY

CopyExplain

While that completes our workflow code review, one more task happens in the background that we don’t have to define. As you may recall, in the first step of the run\_ansible\_playbook job, we had a step that looked for any caches associated with the workflow. Well, by defining that step, there is a post-deploy task that runs at the end of the workflow and creates the cache if one doesn’t exist.

Now that we understand our workflow code, let’s check out a copy of our newly created repository. Copy the code from the example repository and then check in the changes.

Committing the code

As mentioned previously, before running the workflow, we need to check out the empty repository we created at the start of this chapter. This will vary depending on how you interact with GitHub. I use the command line, but you might use the GitHub Desktop application or an IDE such as Visual Studio Code.

Information

For more information on the GitHub desktop application, see https://desktop.github.com/. For details on how to configure an SSH connection to GitHub, see https://docs.github.com/en/authentication/connecting-to-github-with-ssh.

If you want to follow along on the command line, you must update the repository’s name to reflect your own and ensure you have SSH access to your GitHub repositories:

$ git clone https://github.com/PacktPublishing/Learn-Ansible-Second-Edition.git

$ cd Learn-Ansible-Second-Edition-Chapter15

CopyExplain

Once I was in the folder, I copied across the contents of https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter15, ensuring that I also copied the .github folder as this contains the workflow we want to execute.

Once copied, I ran the following commands to add the new files and create the first commit, then pushed:

$ git add .

$ git commit -m "first commit"

$ git push

CopyExplain

If everything goes as planned, if you go to your repository and click on the **Actions** tab, you should see something like this:

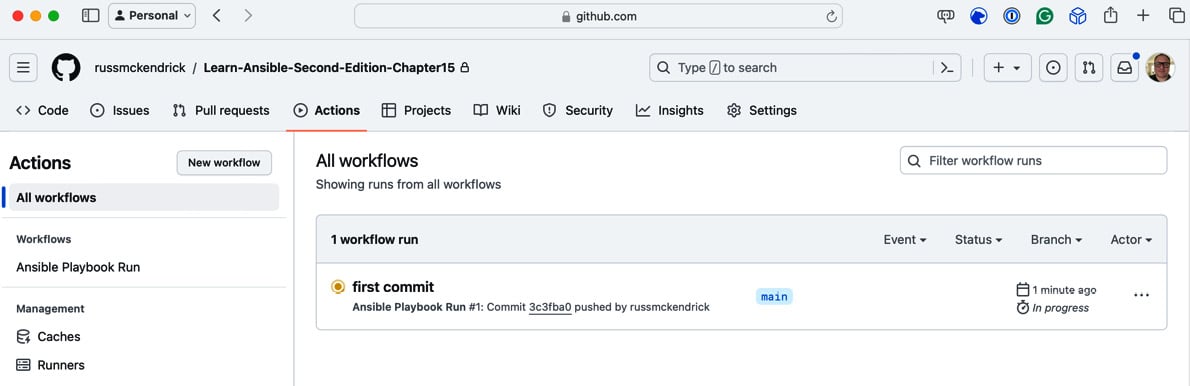


Figure 15.3 – Our first commit is running the GitHub Action

Clicking the name of the commit should show you the progress of the workflow:

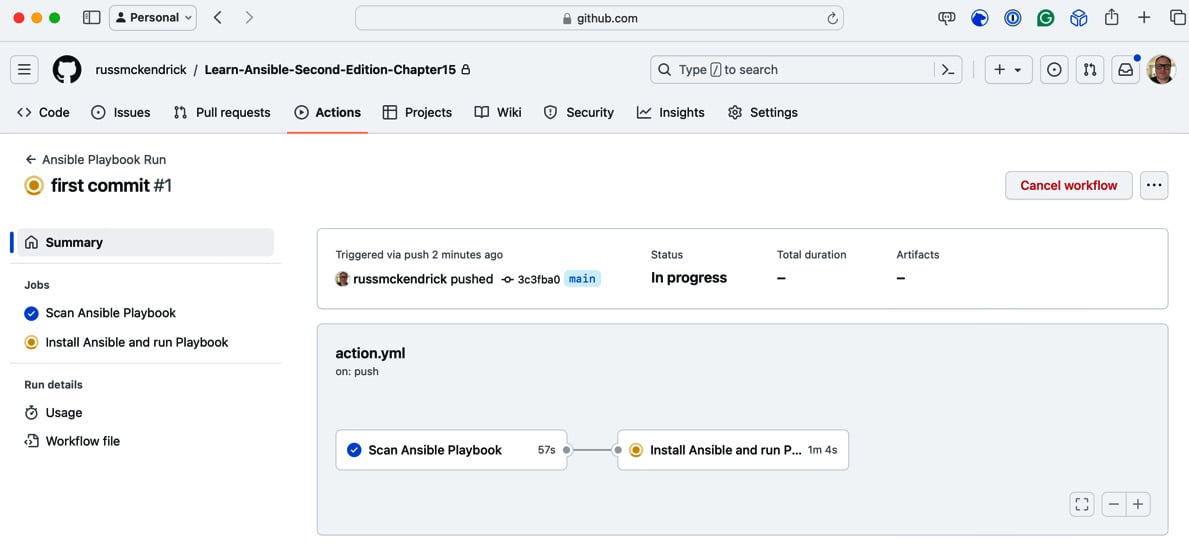


Figure 15.4 – Viewing the progress of the workflow

Click on the running job – in my example, this is the *Install Ansible and Run Playbook* job. This will show you its real-time progress:

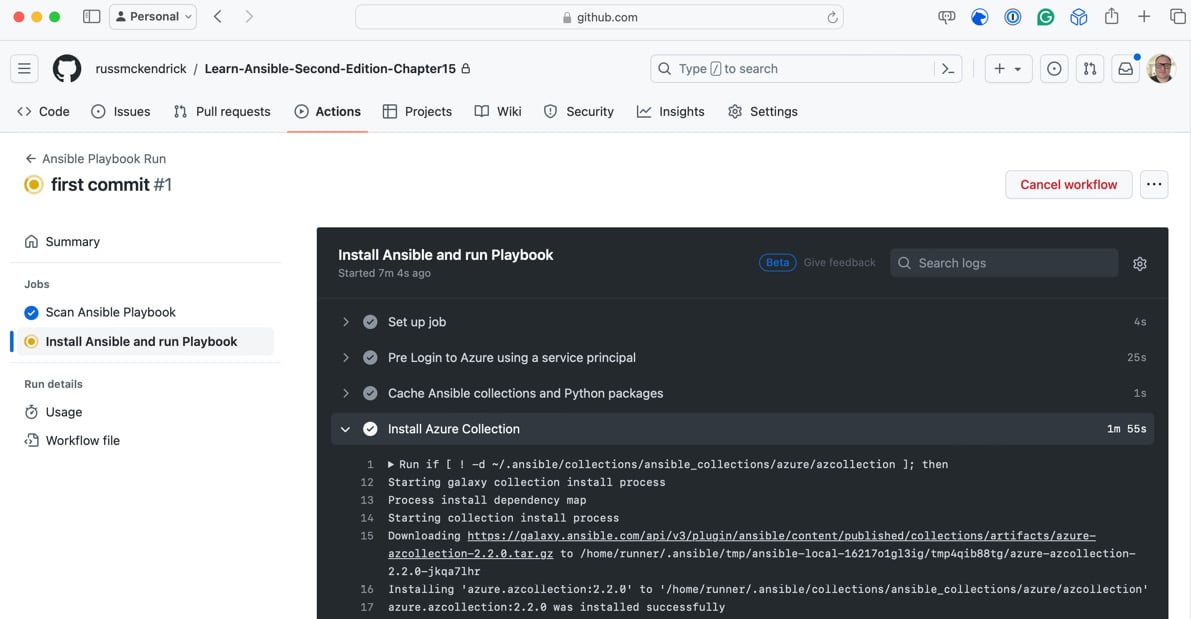


Figure 15.5 – Viewing the real-time output

If everything works as planned, the Ansible playbook will run, the Azure resources will be deployed, and we should have a running WordPress instance.

Clicking on the **Summary** link at the top of the page will show you the full output. Here, we’ll see any warnings or information that was logged during the workflow run, followed by the KICS results:

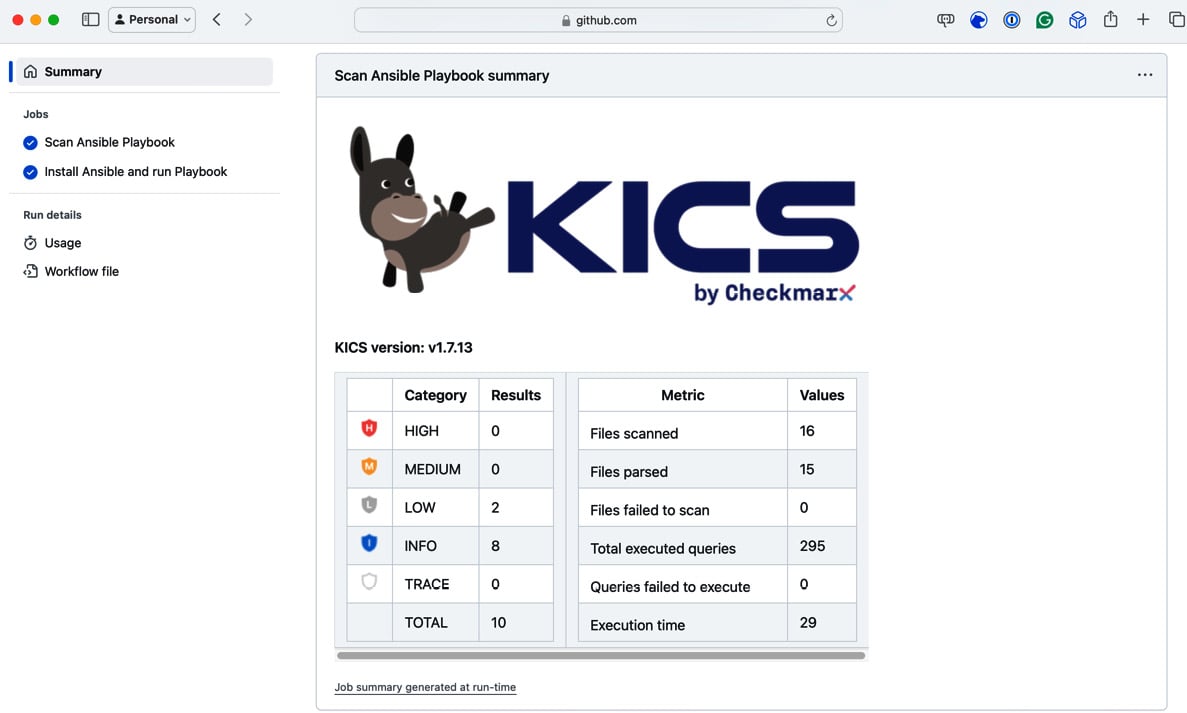


Figure 15.6 – The KICS scan results

You will also be able to expand the **Ansible Playbook Output** area and view the logs:

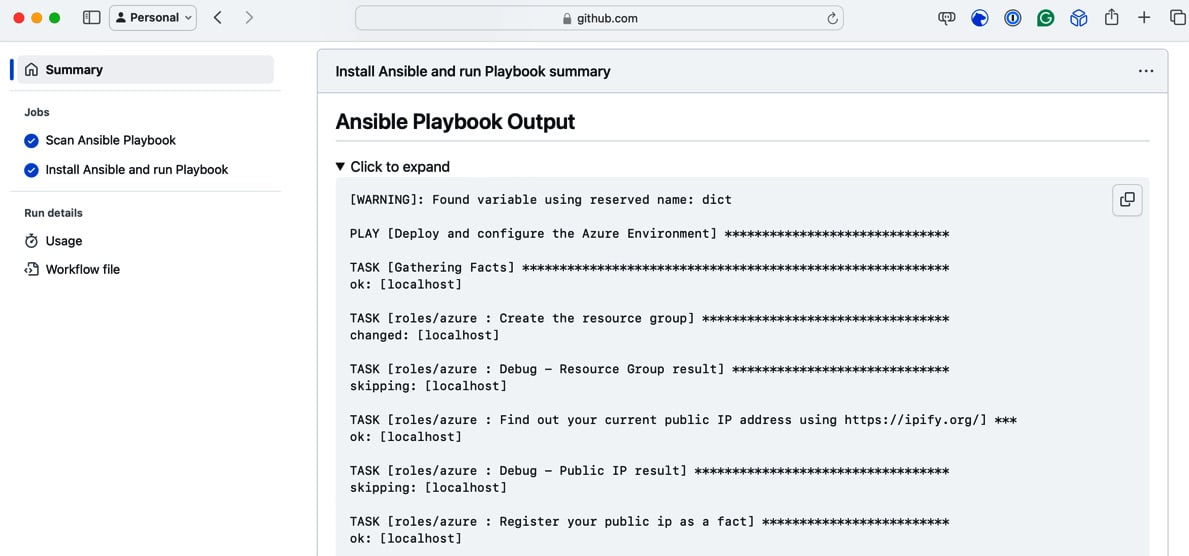


Figure 15.7 – Ansible Playbook Output

Before we remove the Azure resources, let’s see what happens when the scan fails. To do this, open roles/azure/tasks/main.yml and remove the line that reads as follows (it should be around line *61*):

    security\_group: "{{ nsg\_output.state.name }}"

CopyExplain

Once removed, check in the updated code. This will trigger a new workflow run:

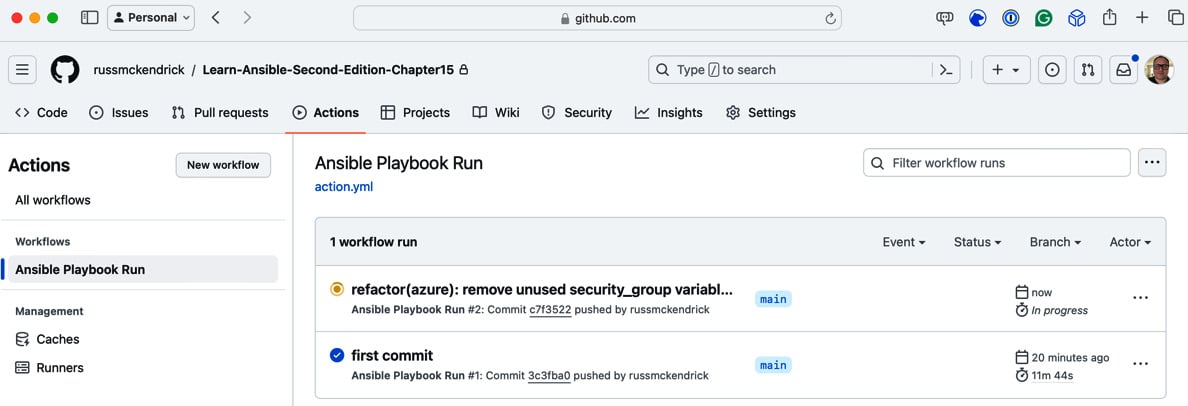


Figure 15.8 – Triggering a second workflow run

As the line we removed will trigger a medium severity rule, our workflow run should fail, as shown here:

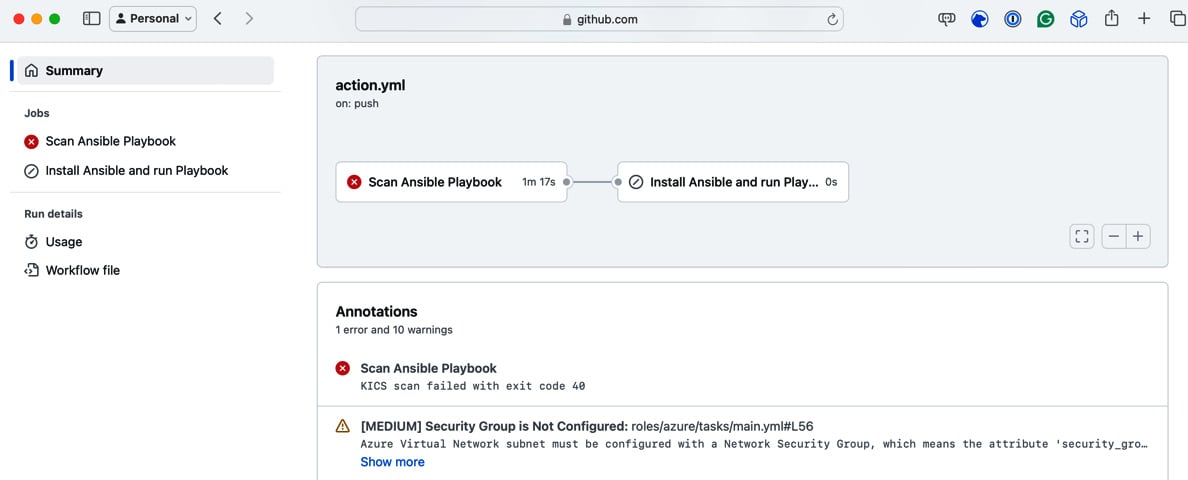


Figure 15.9 – Our second workflow run failed due to our change

Once you have finished testing, I recommend logging into Azure and manually deleting the resource group containing the resources we’ve just launched.

As you can see, while there are considerations you need to make for your deployments – such as ensuring all the connectivity and steps are in place to interact with your cloud provider securely – the general gist and approach to running our Playbooks remains much the same as on our local machine.

The same can also be said about the next tool we will examine, Azure DevOps.

Azure DevOps

The description we used for GitHub Actions also applies to Azure DevOps Pipelines and repositories, two of the Azure DevOps services we will use in this section. Again, we will use platform-provided computing resources to run our Ansible Playbook, and many approaches will be the same. So, rather than covering old ground, let’s start with preparing an Azure DevOps project to host our code and run our Playbook.

Creating and configuring our project

First, you will need to create an Azure DevOps project. Like our GitHub repository, I’ve called it Learn-Ansible-Second-Edition-Chapter15:

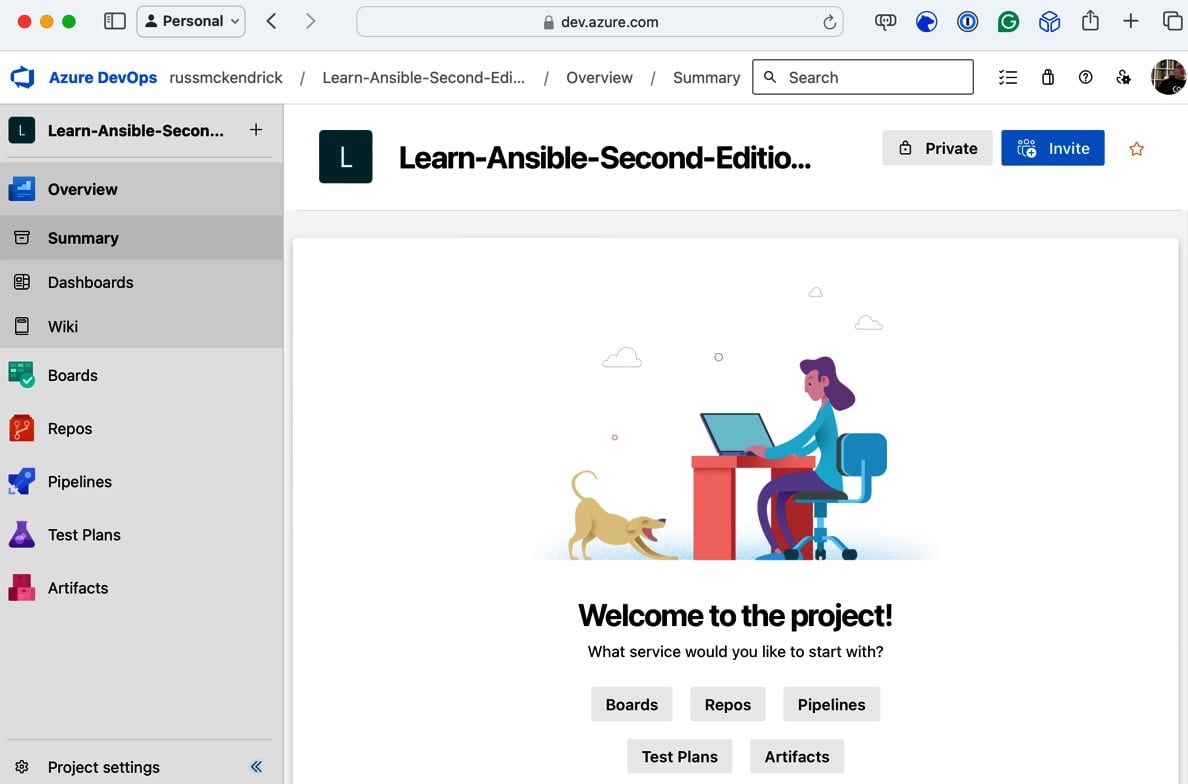


Figure 15.10 – Our newly created Azure DevOps project

We need to configure a few things before checking our code in and adding our pipeline; the first is to create a service connection to Azure itself. To do this, click on the **Project Settings** button, which can be found at the bottom far left-hand corner of the page.

Once **Project Settings** is open, in the left-hand menu under **Pipelines**, click **Service connections**, then click the **Create service connection** button.

Select **Azure Resource Manager**, then click **Next**; from here, select **Service principal (manual)** and click **Next** again.

We are taking this approach rather than any of the others, which would automatically create the service principle for us, as we already have the details of a service principle noted from the *GitHub Actions* section.

The following table contains the information you need to enter:

|  |  |
| --- | --- |
| **Option** | **Content** |
| **Subscription Id** | This is your Azure subscription ID; use the one you granted the service principal access to. In this example, this would be e80d5ad9-e2c5-4ade-a866-bcfbae2b8aea. |
| **Subscription Name** | Enter the name of your Azure subscription. Since we are going to be referring to the subscription ID in the pipeline code, this can be set to anything you like. |
| **Service Principal Id** | This is the appId value from when you create the service principle. In this example, this would be 2616e3df-826d-4d9b-9152-3de141465a69. |
| **Service principal key** | This is the password value that was given when you created the service principle. In this example, this would be Y4j8Q~gVO\*NoTaREalPa55w0rdpP-pdaw. |
| **Tenant ID** | This is the ID of the tenant value listed when you created the service principle. In this example, this would be c5df827f-a940-4d7c-b313-426cb3c6b1fe. |
| **Service connection name** | Enter azConnection here as this is how we are referencing the connection in the pipeline code. |
| **Security** | Ensure that **Grant access permission to all pipelines** is selected. |

Table 15.2 – Information needed for your pipeline in Azure DevOps

Once you’ve entered this information, click the **Verify and Save** button. This will check the details you entered are correct and save the service connection.

Next, we need to install a few extensions from the Visual Studio Marketplace, allowing us to publish our KICS report and an overview of our Playbook run:

* **Markdown Reports**: https://marketplace.visualstudio.com/items?itemName=MasamitsuMurase.publish-markdown-reports
* **Sarif Tools**: https://marketplace.visualstudio.com/items?itemName=sariftools.scans

To enable the extensions on your Azure DevOps organization, follow the preceding URLs and the instructions when you click the **Get it free** button.

The final configuration piece is adding a pipeline variable group and secure file. To do this, click on **Pipelines** in the left-hand side menu and then click **Library**. Once on the **Library** page, click the **+ Variable group** button.

Name the variable group playbook and enter the following variables:

|  |  |
| --- | --- |
| **Name** | **Value** |
| breakSeverity | MEDIUM |
| SSH\_PUBLIC\_KEY | Paste the contents of the id\_rsa.pub file here |
| subscriptionName | azConnection – this is the name of the connection we created at the start of this section |

Table 15.3 – Information required for the variable group

Once you have filled in the preceding information, click **Save**. Once saved, return to the pipeline **Library** area and click **Secure files**; once there, click the **+ Secure file** button and upload the id\_rsa file.

We now have all the base configurations ready and can upload our code.

Cloning the repository and uploading the code

Next, we must clone the repository and upload our code, including the azure-pipelines.yml file we will cover in the next section. To do this, click on **Repos** in the left-hand side menu; you will be presented with several ways to clone the repository.

I’ve chosen to clone using SSH again; if you are following along, update the git clone command to reflect your repository:

$ git clone git@ssh.dev.azure.com:v3/russmckendrick/Learn-Ansible-Second-Edition-Chapter15/Learn-Ansible-Second-Edition-Chapter15

$ cd Learn-Ansible-Second-Edition-Chapter15

CopyExplain

I then copied the files across from https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter15. This time, I didn’t worry about copying the .github directory as it isn’t required. Once the files were in my locally cloned folder, I ran the following commands to add the new files and create the first commit, then push:

$ git add .

$ git commit -m "first commit"

$ git push

CopyExplain

Unlike when we first checked our code into GitHub, nothing will happen because we haven’t configured our pipeline yet.

The Azure DevOps pipeline

Our pipeline is defined in the azure-pipelines.yml file, which can be found at the root of our repository file. Let’s quickly review the content before we create the pipeline using that file.

Information

Structurally, our azure-pipelines.yml file is close to what we have already covered for GitHub Actions; in fact, you might almost think they are interchangeable and compatible – however, they aren’t, so please be careful not to mix the two up.

Our pipeline file starts with a basic configuration that instructs the pipeline when to trigger, which variable group to load, and which underlying image to use. Right at the top, there’s an exclusion rule for KICS, something we covered in [*Chapter 13*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/13), *Scanning Your Ansible Playbooks*:

# kics-scan disable=3e2d3b2f-c22a-4df1-9cc6-a7a0aebb0c99

trigger:

  - main

variables:

  - group: playbook

pool:

  vmImage: ubuntu-latest

CopyExplain

Once the basic configuration is complete, we can start the stages:

1. Our first run is the KICS scan on the code:
2. - stage: "scan"

    displayName: "KICS - Scan Ansible Playbook"

CopyExplain

1. This stage is made up of a single job:
2. jobs:
3. - job: "kics\_scan"
4. displayName: "Run KICS Scan"
5. pool:
6. vmImage: "ubuntu-latest"

        container: checkmarx/kics:debian

CopyExplain

1. As you may have noticed, here, we are using the checkmarx/kics:debian container image to deploy KICS. This will spin up the container and run the following steps from within it. Our step contains two tasks – the first creates the output folder, checks out the code, and runs the scan:
2. steps:
3. - script: |
4. mkdir -p $(System.DefaultWorkingDirectory)/output
5. /app/bin/kics scan --ci -p ${PWD} -o ${PWD} --report-formats "all" --ignore-on-exit results
6. mv results\* $(System.DefaultWorkingDirectory)/output

              ls -lhat $(System.DefaultWorkingDirectory)/output

CopyExplain

1. The second task publishes the content of the output directory, which contains all of our scan results as a build artifact:
2. - task: PublishBuildArtifacts@1
3. inputs:
4. pathToPublish: $(System.DefaultWorkingDirectory)/output

              artifactName: CodeAnalysisLogs

CopyExplain

1. With the files published, we no longer need the resources that were generated during this stage, so we can move on to the second stage:
2. - stage: "scan\_parse"
3. displayName: "KICS - Parse Scan Resaults"
4. jobs:
5. - job: "kics\_scan\_parse\_result"
6. displayName: "Check KICS Scan Resaults"
7. pool:
8. vmImage: "ubuntu-latest"

        steps:

CopyExplain

1. As you can see, this stage parses our scan results; the first task we run downloads a copy of the artifact we uploaded during the last stage:
2. - task: DownloadPipelineArtifact@2
3. displayName: "Download the Security Scan Artifact Result"
4. inputs:

              artifact: CodeAnalysisLogs

CopyExplain

Now that we have the results files, we need to review them to figure out if the Ansible Playbook should be run or not. This task runs a bash script that reads the JSON results and sets some pipeline variables to control what happens next.

1. We start the task with some configuration:
2. - task: Bash@3
3. name: "setvar"
4. displayName: "Check for issues in the scan result"
5. inputs:
6. failOnStderr: true
7. targetType: "inline"

                script: |

CopyExplain

1. Now, we have the script itself, which starts by setting some local variables and printing some results out to the screen using the echo command. These will appear in our pipeline run:
2. resultsFilePath="$(Pipeline.Workspace)/results.json"
3. BREAK=$(breakSeverity)
4. echo "Checking for severity level: $BREAK"
5. noIssues=$(jq --arg BREAK "$BREAK" '.severity\_counters[$BREAK] // 0' $resultsFilePath)

                  echo "Number of issues found: $noIssues"

CopyExplain

Then, we create a *group*, which means that when we review the pipeline output, the following information will be minimized, making it easier to read.

1. In the group, we have an if statement that states that if less than (-lt) 1 issues are detected (that is, zero issues), then the output variable, OK\_TO\_DEPLOY, is set to true:
2. echo "##[group]Checking the scan output"
3. if [ "$noIssues" -lt 1 ]; then
4. echo "##vso[task.setvariable variable=OK\_TO\_DEPLOY;isOutput=true]true"

                      echo "##vso[task.logissue type=warning]No issue found. Progressing with pipeline."

CopyExplain

1. If this condition is not met – that is, there are one or more issues – then OK\_TO\_DEPLOY is set to false and an error is logged:
2. else
3. echo "##vso[task.setvariable variable=OK\_TO\_DEPLOY;isOutput=true]false"
4. echo "##vso[task.logissue type=error]Pipeline failed due to $noIssues issue(s) found."
5. fi

                  echo "##[endgroup]"

CopyExplain

1. Logging the error will stop the remainder of the pipeline from running. The next and final stage runs the Ansible Playbook. It has a dependency on the previous stage being successfully executed and OK\_TO\_DEPLOY being set to true:
2. - stage: "run\_ansible"
3. displayName: "Run Ansible"
4. condition: |
5. and
6. (
7. succeeded(),
8. eq(dependencies.scan\_parse.outputs['kics\_scan\_parse\_result.setvar.OK\_TO\_DEPLOY'], 'true')
9. )
10. jobs:
11. - job: "ansible\_install"
12. displayName: "Ansible"

        steps:

CopyExplain

1. The first task logs us into Azure and sets the service principle details as environment variables for use in a later task:
2. - task: AzureCLI@2
3. displayName: 'Azure CLI'
4. inputs:
5. azureSubscription: '$(subscriptionName)'
6. addSpnToEnvironment: true
7. scriptType: 'bash'
8. scriptLocation: 'inlineScript'
9. inlineScript: |
10. echo "##vso[task.setvariable variable=ARM\_SUBSCRIPTION\_ID]$(az account show --query="id" -o tsv)"
11. echo "##vso[task.setvariable variable=ARM\_CLIENT\_ID]${servicePrincipalId}"
12. echo "##vso[task.setvariable variable=ARM\_CLIENT\_SECRET]${servicePrincipalKey}"

                echo "##vso[task.setvariable variable=ARM\_TENANT\_ID]${tenantId}"

CopyExplain

1. Next up, we need to add our SSH key to our environment. This uses the secure file we uploaded earlier:
2. - task: InstallSSHKey@0
3. displayName: "Add SSH Key"
4. inputs:
5. sshKeySecureFile: "id\_rsa"

              knownHostsEntry: "azure.devops"

CopyExplain

1. Now, we need to add the public portion of the SSH key, install the bits we need to run the Ansible Playbook, and then actually run it, remembering to add the details for the service principle:
2. - task: Bash@3
3. name: "ansible"
4. displayName: "Run Ansible"
5. env:
6. AZURE\_CLIENT\_ID: $(ARM\_CLIENT\_ID)
7. AZURE\_SECRET: $(ARM\_CLIENT\_SECRET)
8. AZURE\_TENANT: $(ARM\_TENANT\_ID)
9. AZURE\_SUBSCRIPTION\_ID: $(ARM\_SUBSCRIPTION\_ID)
10. ANSIBLE\_HOST\_KEY\_CHECKING: "False"
11. inputs:
12. targetType: "inline"

                script: |

CopyExplain

1. With the environment ready, we can run the script, which starts by adding the id\_rsa.pub file and adding the right permissions:
2. echo "##[group]Add SSH key"
3. echo "$(SSH\_PUBLIC\_KEY)" > ~/.ssh/id\_rsa.pub
4. chmod 644 ~/.ssh/id\_rsa.pub

                  echo "##[endgroup]"

CopyExplain

1. The next part of the script installs the Azure Ansible collection from Ansible Galaxy and installs the requirements. We are using --force here to ensure that the latest copy of all the collection is pulled down from Ansible Galaxy:
2. echo "##[group]Install the Azure Ansible Collection"
3. ansible-galaxy collection install --force azure.azcollection
4. pip3 install -r ~/.ansible/collections/ansible\_collections/azure/azcollection/requirements-azure.txt

                  echo "##[endgroup]"

CopyExplain

1. With those installed, we can now run the playbook; we are taking a similar approach to running the playbook as we did for our GitHub Action:
2. echo "##[group]Run the Ansible Playbook"
3. ansible-playbook -i inv site.yml 2>&1 | tee $(System.DefaultWorkingDirectory)/ansible\_output.log

                  echo "##[endgroup]"

CopyExplain

1. The final part of our script takes our Ansible output and creates a Markdown file called summary.md:
2. echo "##[group]Create the mardown file for the Ansible Playbook Output"
3. mkdir -p $(System.DefaultWorkingDirectory)/markdown
4. echo "# Ansible Playbook Output" > $(System.DefaultWorkingDirectory)/markdown/summary.md
5. echo "<details><summary>Click to expand</summary>" >> $(System.DefaultWorkingDirectory)/markdown/summary.md
6. echo "" >> $(System.DefaultWorkingDirectory)/markdown/summary.md
7. echo "\`\`\`" >> $(System.DefaultWorkingDirectory)/markdown/summary.md
8. cat $(System.DefaultWorkingDirectory)/ansible\_output.log >> $(System.DefaultWorkingDirectory)/markdown/summary.md
9. echo "\`\`\`" >> $(System.DefaultWorkingDirectory)/markdown/summary.md                      echo "</details>" >> $(System.DefaultWorkingDirectory)/markdown/summary.md

                  echo "##[endgroup]"

CopyExplain

1. The final task of the pipeline is to upload a copy of the markdown/summary.md file to our pipeline:
2. - task: PublishMarkdownReports@1
3. name: "upload\_ansible\_output"
4. displayName: "Upload Ansible Output"
5. inputs:
6. contentPath: "$(Build.SourcesDirectory)/markdown"

              indexFile: "summary.md"

CopyExplain

With that, our pipeline is complete. So, now that we know what it does, let’s add it to our Azure DevOps project and run it for the first time.

If you click on **Pipelines** in the left-hand side menu and then click the **Create Pipeline** button, you will be asked, **Where is your code?**. select **Azure Repos Git**, and then your repository – the azure-pipelines.yml file will be loaded and you will have the option to **Run** or **Save**. We’ll click **Run**.

You will be presented with something like the following screen:

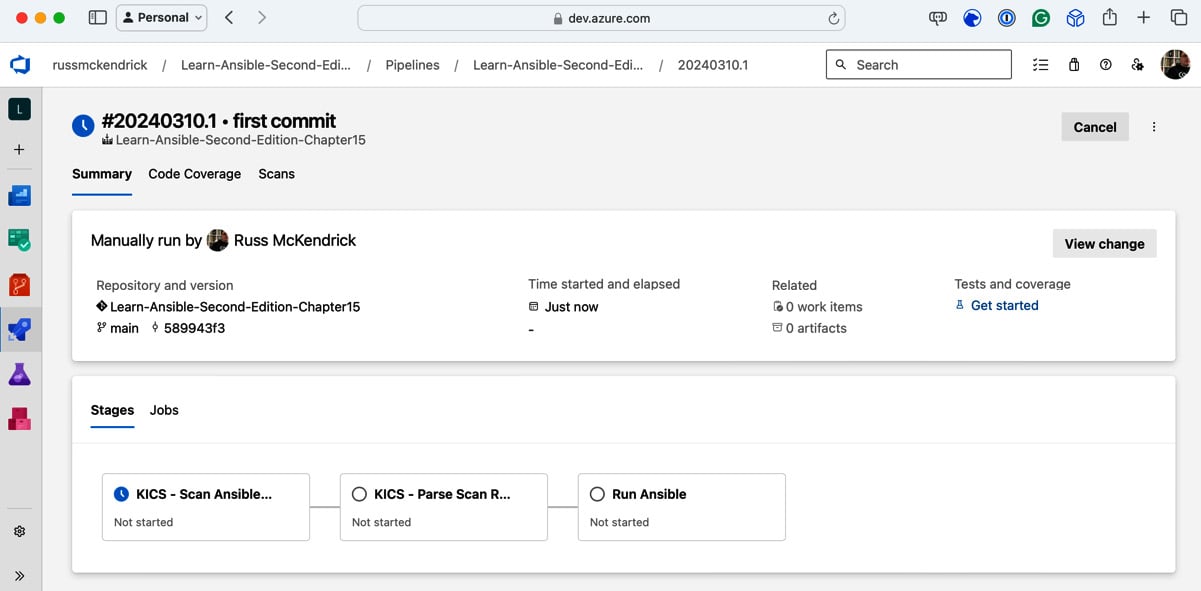


Figure 15.11 – Running the pipeline for the first time

However, not is all as it seems! If you click on the first stage, you will be presented with the following. The pipeline needs permissions to access the variable group we created:

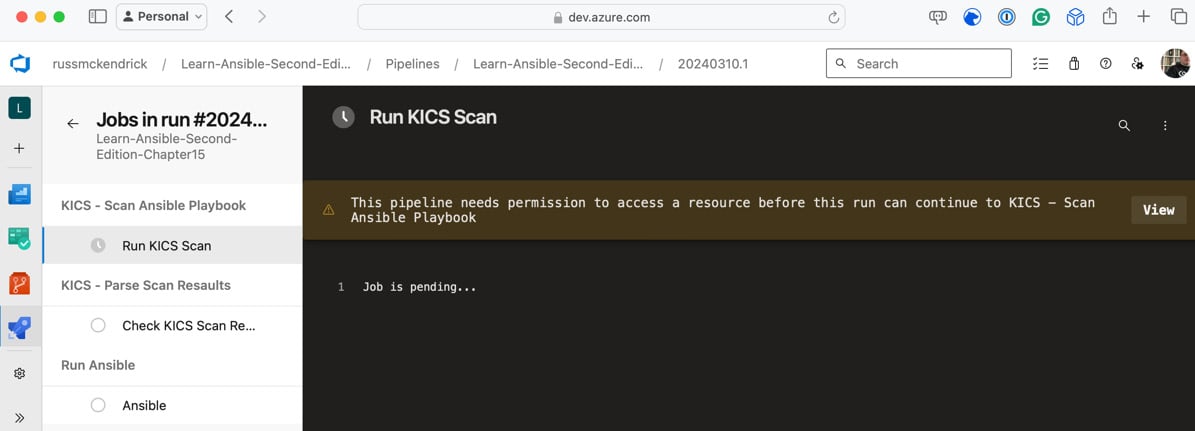


Figure 15.12 – Granting the permissions for the variable group

Click **View** and follow the onscreen instructions to grant the permissions. The KICS scan will run, and the stage will be complete. It will then move on to the **Parse Scan Results** stage, which should be completed again.

If you go back to the summary, you’ll see that more permissions are required, this time to access the secure file we uploaded:

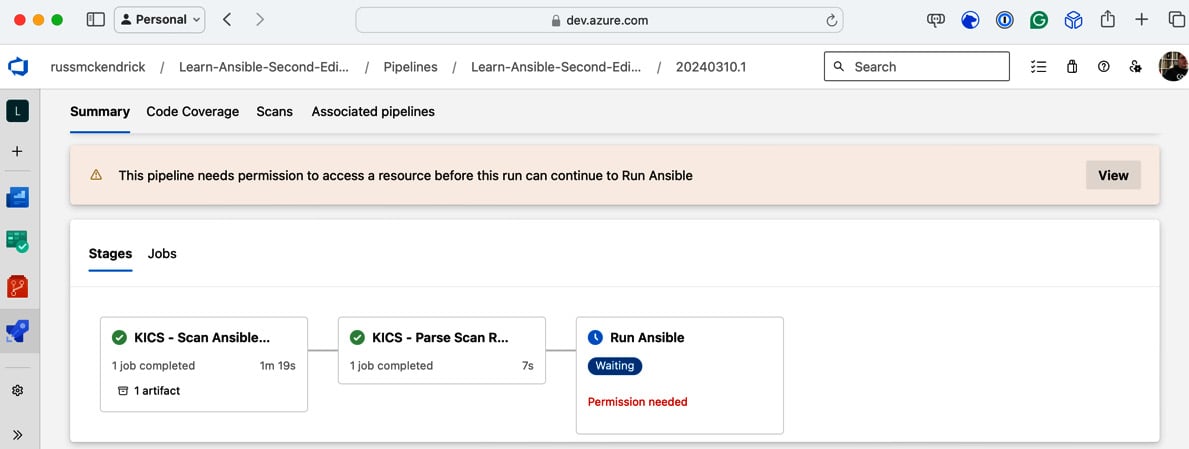


Figure 15.13 – Grant the permissions for the secure file

Again, click **View** and follow the onscreen instructions to grant permission. This should be the last permission that needs to be given. From now on, when we run the pipeline, permissions will already be given.

If you click on the **Run Ansible** stage, you can keep track of the Playbook run. If everything goes as planned, returning to the summary should show you something like the following:

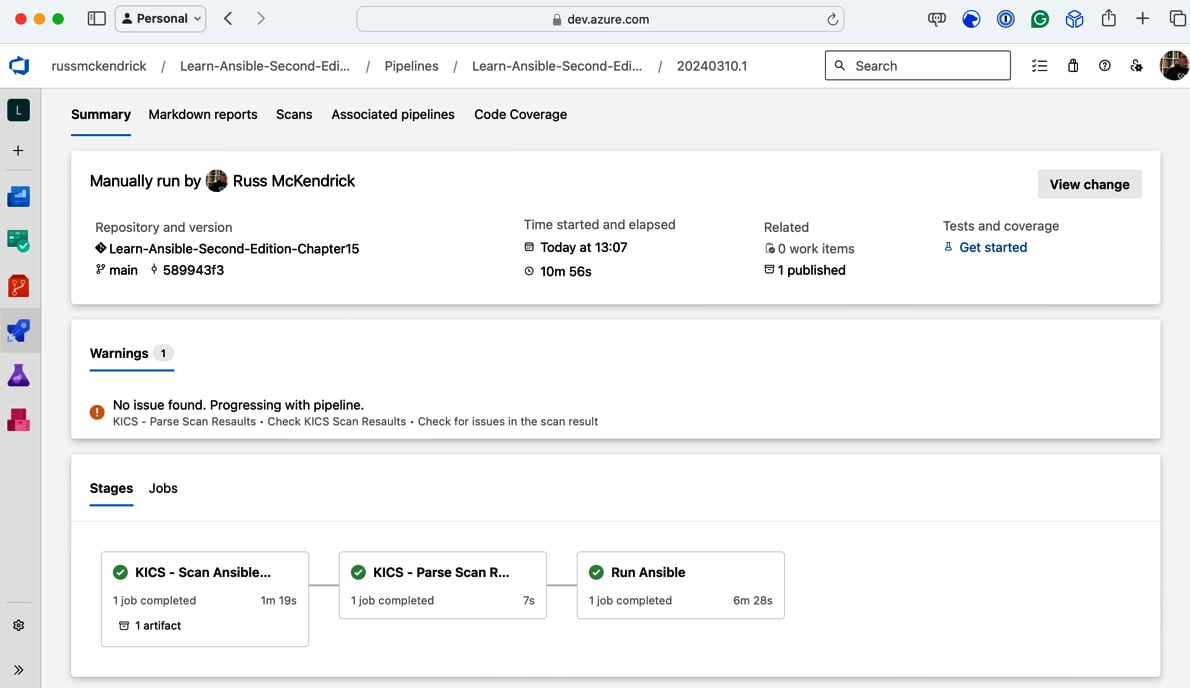


Figure 15.14 – Everything worked!!!

Clicking on **Markdown reports** will show the result of the Playbook run:

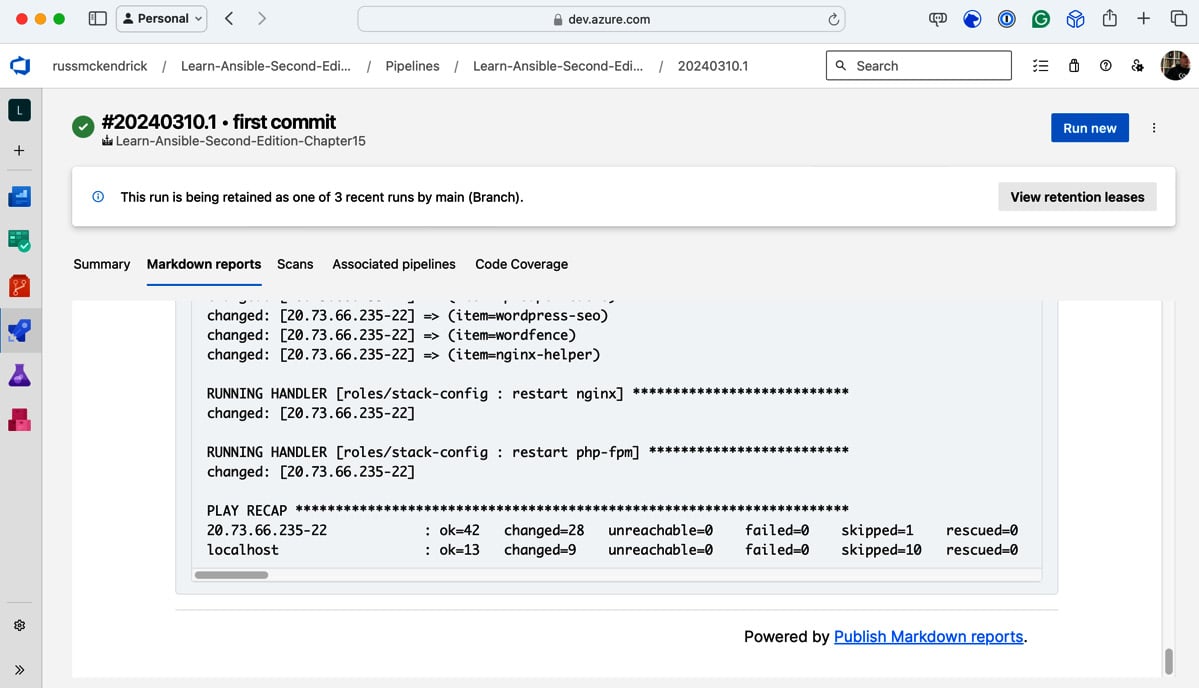


Figure 15.15 – The Markdown report

Clicking **Scans** will show you the results of the KICS scan:

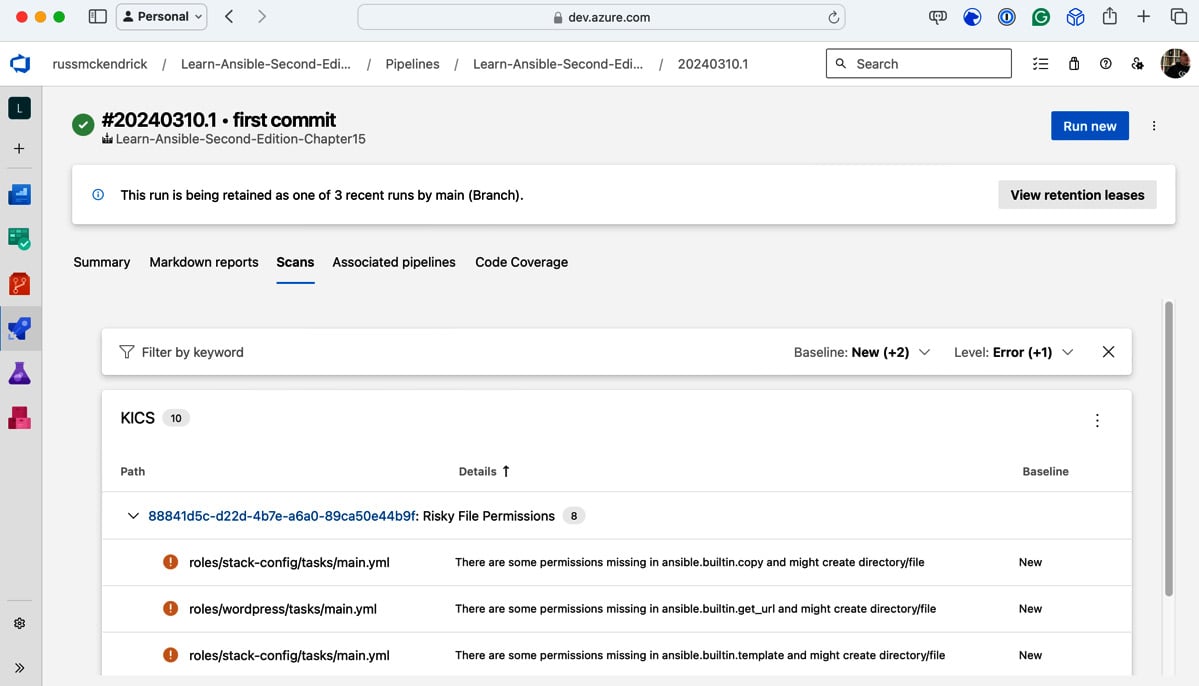


Figure 15.16 – The scan report

Like GitHub Actions, let’s see what happens when the scan fails. Again, open roles/azure/tasks/main.yml and remove the line that reads as follows (it should be around line *61*):

    security\_group: "{{ nsg\_output.state.name }}"

CopyExplain

Once removed, check in the updated code. This will trigger a new workflow run:

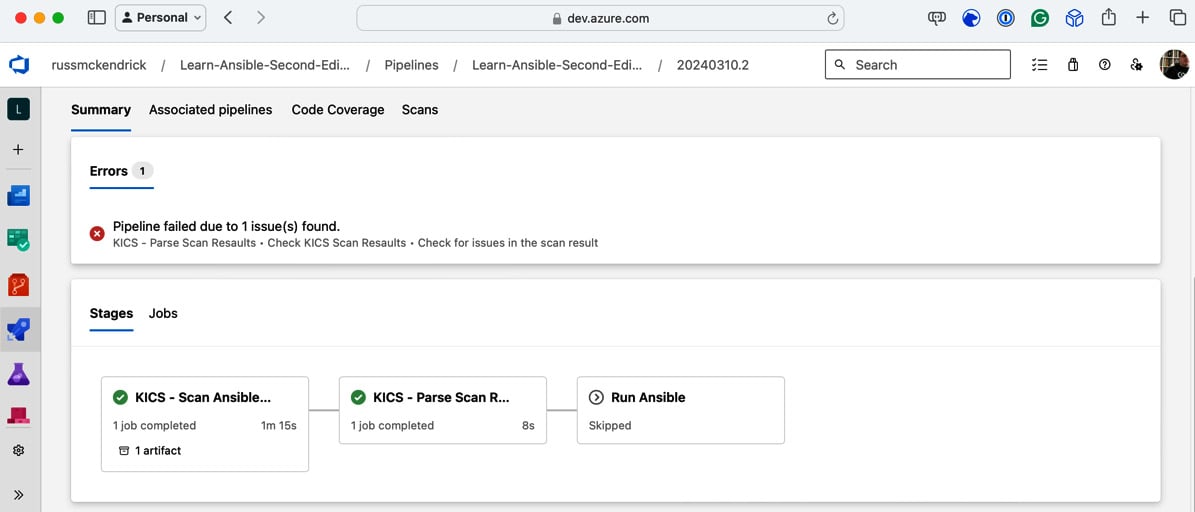


Figure 15.17 – The pipeline has errored

As you can see, we have a message stating **Pipeline failed due to 1 issue(s) found**, and the **Run Ansible** stage was skipped as we didn’t meet the conditions for it to run.

Once you have finished testing, log into Azure and manually delete the resource group containing the resources we have just launched.

Summary

In this chapter, we looked at running our Ansible Playbooks using the compute resources GitHub and Azure DevOps provide. We discovered that this is great for running our playbook code as we can ship code that defines the configuration for the computing resources alongside our Playbook code.

We also learned that by using the built-in tools, we can securely configure our environment so as not to expose secrets, such as our service principle credentials, to other users who have access to run the playbook.

The only downside is that we had to create the logic that runs the playbook. Wouldn’t it be great to use a tool designed to centrally run our Playbooks from a single user interface? Well, in our next chapter, we will cover exactly that – so if you like the approach we have taken so far, read on.