15. Continuous integration and continuous deployment for AKS

DevOps is the union of people, processes, and tools to deliver software faster, more frequently, and more reliably. Within the DevOps culture are the practices of **continuous integration and continuous deployment** (**CI/CD**). CI/CD is a set of practices, implemented through one or more tools, to automatically test, build, and deliver software.

The CI phase refers to the practice of continuously testing and building software. The outcome of the CI phase is a deployable artifact. That artifact could be many things; for instance, for a Java application it would be a `JAR` file, and in the case of a container-based application it would be a container image.

The CD phase refers to the practice of continuously releasing software. During the CD phase, the artifact that was generated during CI is de-

ployed to multiple environments, typically going from test to QA to staging to production.

Multiple tools exist to implement CI/CD. GitHub Actions is one such tool. GitHub Actions is a workflow automation system built into GitHub. With GitHub Actions, you can build, test, and deploy applications written in any language to a variety of platforms. It also allows you to build container images and deploy applications to a Kubernetes cluster, which you'll do in this chapter.

Specifically, this chapter will cover the following topics:

- CI/CD process for containers and Kubernetes
- Setting up Azure and GitHub
- Setting up a CI pipeline
- Setting up a CD pipeline

Let's start by exploring the CI/CD lifecycle for containers and Kubernetes.

CI/CD process for containers and Kubernetes

Before you start building a pipeline, it's good to understand the typical CI/CD process for containers and Kubernetes. In this section, the high-level process shown in *Figure 15.1* will be explored in more depth. For a more detailed exploration on CI/CD and DevOps for Kubernetes, you are encouraged to explore the following free online eBook by Microsoft: **https://docs.microsoft.com/dotnet/architecture/containerized-lifecycle/**.

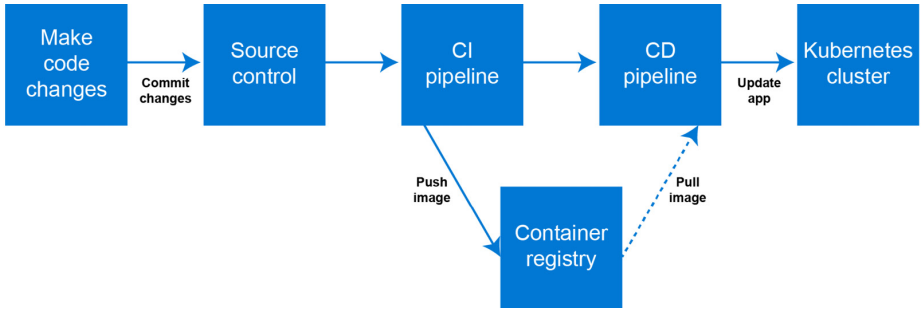

Figure 15.1: Container and Kubernetes CI/CD process

The process starts with somebody making code changes. Code changes could mean application code changes, changes to the Dockerfile used to build the container, or changes to the Kubernetes YAML files used to deploy the application on a cluster.

Once code changes are complete, those changes are committed to a source control system. Typically, this is a Git repository, but other systems, such as Subversion (SVN), also exist. In a Git repository, you would usually have multiple branches of your code. Branches enable multiple individuals and teams to work on the same code base in parallel without interfering with each other. Once the work done on a branch is complete, it is merged with the main (or master) branch. Once a branch is merged, the changes from that branch are shared with others using that code base.

Note

Branches are a powerful functionality of the Git source control system. There are multiple ways to manage how you use branches in a code base. Please refer to the chapter on branches in Scott Chacon and Ben Straub's Pro Git (Apress, 2014) for a more in-depth exploration of this topic: **https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell**.

After code is pushed into source control, either in the main branch or a feature branch, a CI

pipeline can be triggered. In a container-based application, this means that the code is built into a container image, that image is tested, and if tests succeed, it is pushed to a container registry. Depending on the branch, you could include different steps and different tests. For example, on feature branches you might only build and test the container to verify the code works but not push it to a registry, while on the main branch you might build and test the container and push it to a container registry.

Finally, a CD pipeline can be triggered to deploy or update your application on Kubernetes. Typically, in a CD pipeline, the deployment moves through different stages. You can deploy your updated application first to a staging environment, where you can run both automated and manual tests on the application before moving it to production.

Now that you've got an understanding of the CI/CD process for containers and Kubernetes, you can start building the example part of this chapter. Let's start with setting up Azure and GitHub to do this.

Setting up Azure and GitHub

In this section, you'll set up the basic infrastructure you'll use to create and run the pipeline that you will build. To host your container images, you need a container registry. You could use a number of container registries, but here you'll create an Azure Container Registry instance because it is well integrated with **Azure Kubernetes Service** (**AKS**). After creating the container registry, you will need to link that container registry to your AKS cluster and create a new service principal, and then you'll need to set up a GitHub repository to run the example part of this chapter. Execute the following seven steps to complete this activity:

1. To start, create a new container registry. In the Azure search bar, look for **`container registry`** and click on Container registries, as shown in *Figure 15.2:*

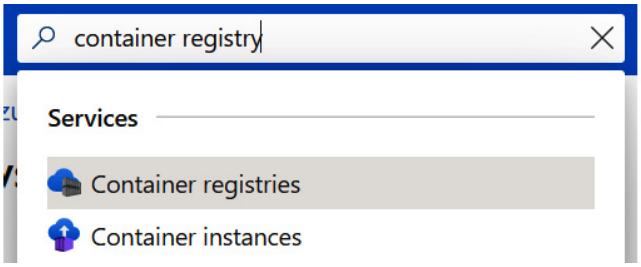

Figure 15.2: Navigating to the Container registry service through the Azure portal

2. Click the Create button at the top to create a new registry. To organize the resources in this chapter together, create a new resource group. To do this, click on Create new to create a new resource group and call it **`rg-pipelines`**, as shown in *Figure 15.3:*

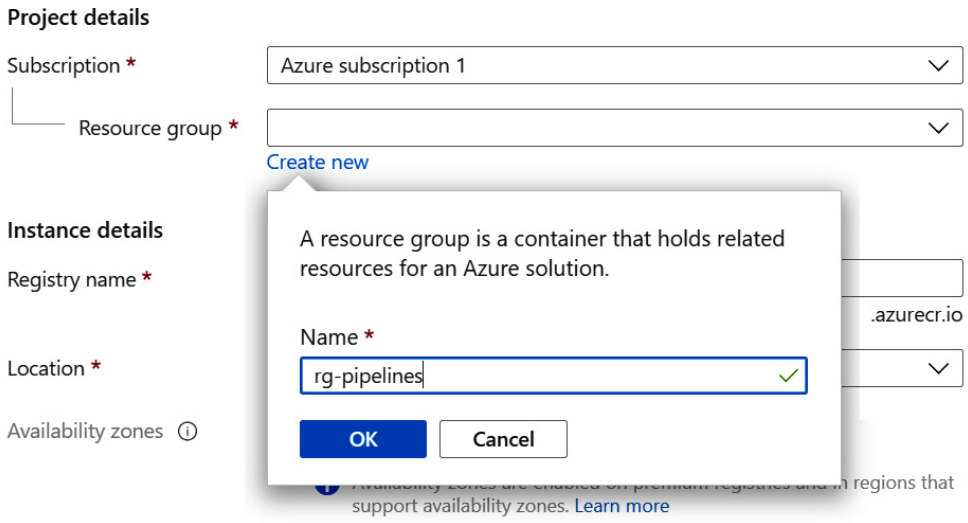

Figure 15.3: Creating a new resource group

Provide the details required to create the registry. The registry name needs to be globally unique, so consider adding your initials to the registry name. It is recommended to create the registry in the same location as your cluster. To optimize the spend for the demo, you can change the SKU to Basic. Select the Review +

Create button at the bottom to create the registry, as shown in *Figure 15.4:*

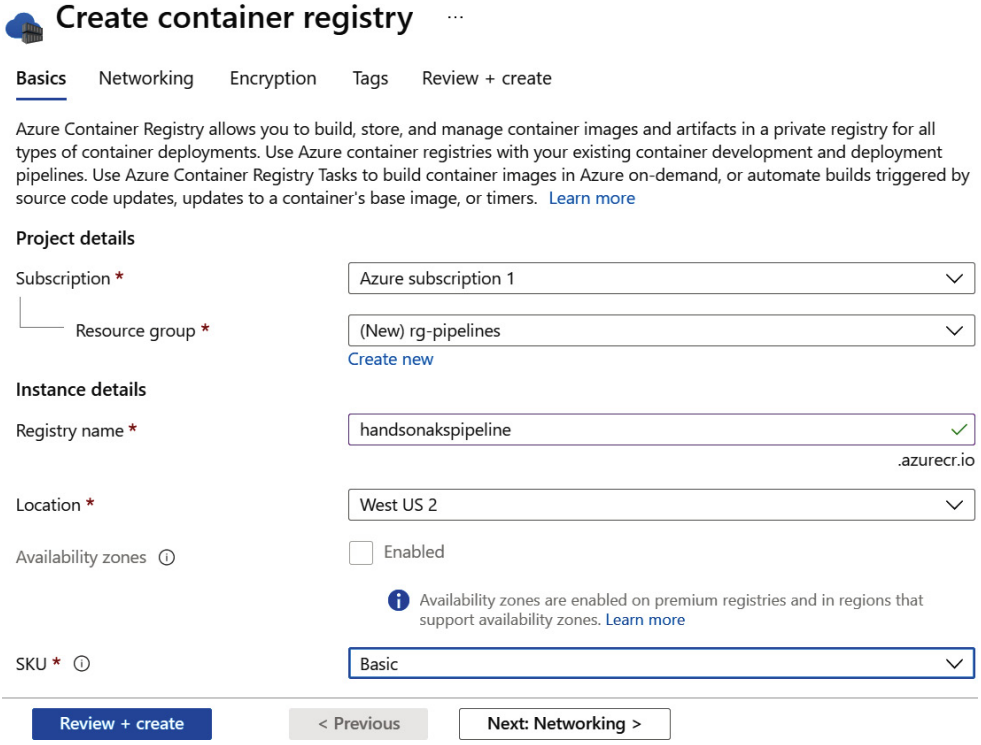

Figure 15.4: Creating a new container registry

In the resulting pane, click the Create button to create the registry.

3. When your registry is created, open Cloud Shell so that you can configure your AKS cluster to get access to your container registry. Use the following command to give AKS permissions on your registry:

   az aks update -n handsonaks \
   -g rg-handsonaks --attach-acr <acrName>

   This will return an output similar to *Figure 15.5*, which has been cropped to show only the

top part of the output:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter15$ az aks update -n handsonaks \
>  -g rg-handsonaks --attach-acr handsonakspipeline
The behavior of this command has been altered by the following extension: aks-preview
AAD role propagation done[############################################]  100.0000%{
  "aadProfile": null,
  "addonProfiles": {
    "KubeDashboard": {
      "config": null,
      "enabled": false,
      "identity": null
    },
    "azurepolicy": {
      "config": {
        "version": "v2"
      },
```

Figure 15.5: Allowing AKS cluster to access the container registry

4. Next, you'll need to create a service principal that will be used by GitHub Actions to connect to your subscription. You can create this service principal using the following command:

   az ad sp create-for-rbac --name "cicd-pipeline" \

   --sdk-auth --role contributor

   You will need the full output JSON of this command, as highlighted in *Figure 15.6*, later in GitHub. Copy this output:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter15$ az ad sp create-for-rbac --name "cicd-pipeline" \
> --sdk-auth --role contributor
Changing "cicd-pipeline" to a valid URI of "http://cicd-pipeline", which is the required format used for
 service principal names
Creating 'contributor' role assignment under scope '/subscriptions/ede7a1e5-4121-427f-876e-e100eba989a0'
  Retrying role assignment creation: 1/36
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.m
s/azadsp-cli
{
  "clientId": "a66da355-58a2-410d-af4c-f9cfe44f7348",
  "clientSecret": "NtMhnqeY4Iw-TrSDk~NLv-P-pdsMdn_IYf",
  "subscriptionId": "ede7a1e5-4121-427f-876e-e100eba989a0",
  "tenantId": "1cf4b872-ae04-44c8-8318-2ba43e95f591",
  "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
  "resourceManagerEndpointUrl": "https://management.azure.com/",
  "activeDirectoryGraphResourceId": "https://graph.windows.net/",
  "sqlManagementEndpointUrl": "https://management.core.windows.net:8443/",
  "galleryEndpointUrl": "https://gallery.azure.com/",
  "managementEndpointUrl": "https://management.core.windows.net/"
}
```

Figure 15.6: Creating a new service principal

5. This completes the Azure part of the setup. Next, you'll need to log in to GitHub, fork the repo that comes with this book, and configure a secret in this repo. If you do not yet have a GitHub account, please create one via **https://github.com/join**. If you already have an account, please sign in using **https://github.com/login**.
6. Once you are logged in to GitHub, browse to the repository associated with this book at **https://github.com/PacktPublishing/Hands-On-Kubernetes-on-Azure-third-edition**. Create a fork of this repo in your account by clicking on the Fork button in the top-right corner of the screen, as shown in *Figure 15.7:*

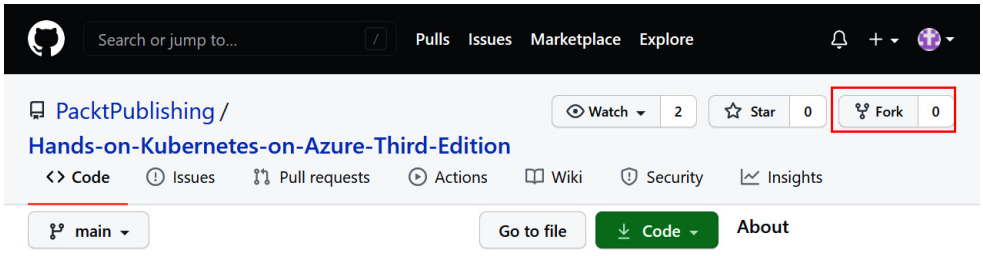

Figure 15.7: Forking the GitHub repository

Forking the repo will create a copy of the repository in your own GitHub account. This will allow you to make changes to the reposi-

tory, as you will do as you build the pipeline in this chapter.

7. Forking the repository takes a couple of seconds. Once you have the fork in your own account, you'll need to configure the Azure secret in this repo. Start by clicking on Settings in the top-right corner of your repo, as shown in *Figure 15.8:*

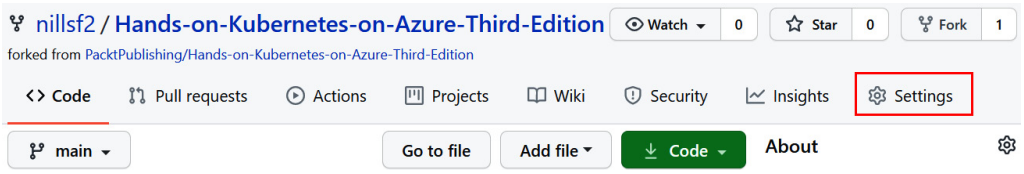

Figure 15.8: Clicking on settings in the GitHub repository

This will take you to the setting of your repo. On the left-hand side, click on Secrets, and on the resulting screen click on the New repository secret button at the top, as shown in *Figure 15.9:*

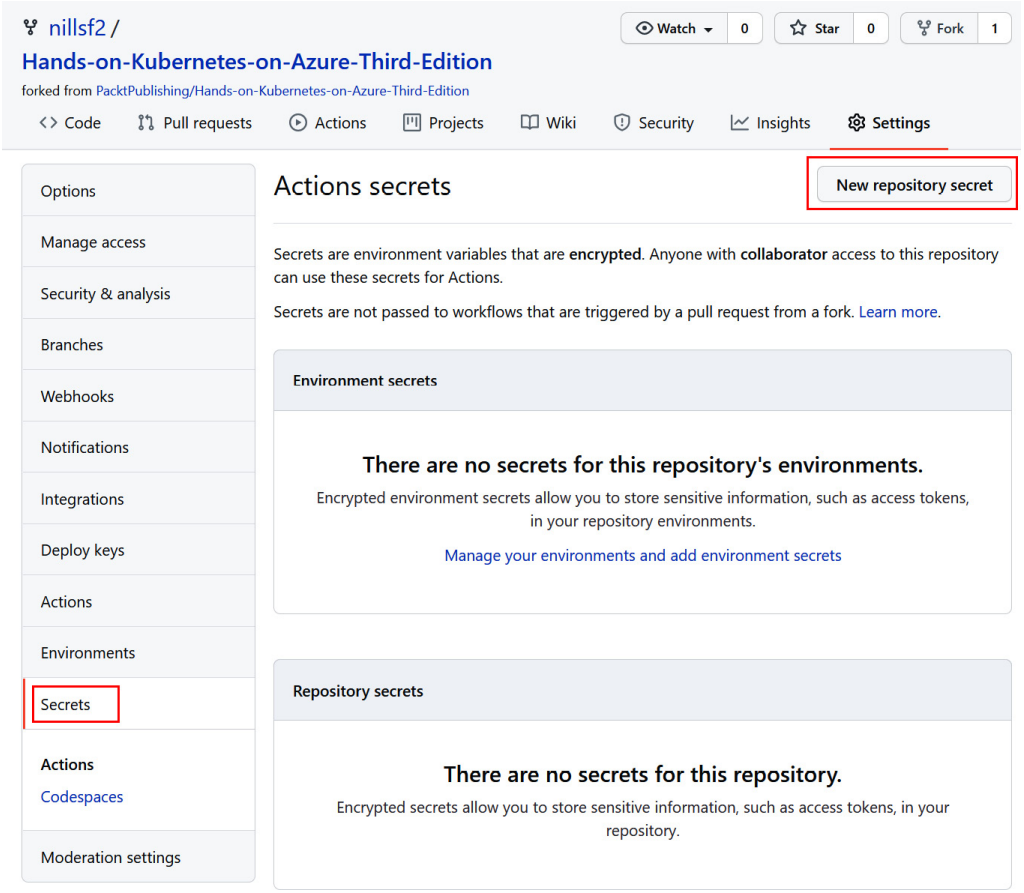

Figure 15.9: Creating a new repository secret

This will take you to the screen to create the new secret. Call this secret **`AZURE_CREDENTIALS`**, and as the value for the secret, paste in the output from the CLI command you issued in *step 4* of this section, as shown in *Figure 15.10:*

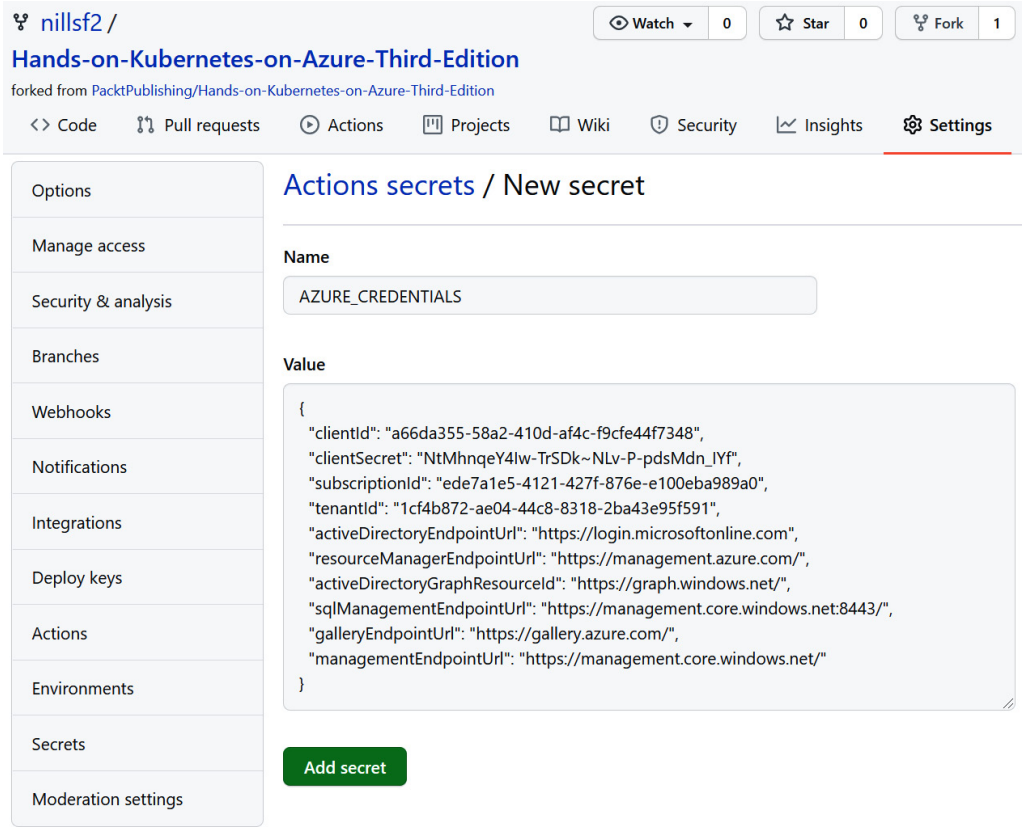

Figure 15.10: Setting of the value of the new secret

Finally, click on Add secret at the bottom of this screen to save the secret.

Now you have set up Azure and GitHub to start building your pipeline. You have created a service principal that GitHub will use to interact with Azure, and you created a container registry that your CI pipeline can push images to and that AKS can pull images from. Let's now build a CI pipeline.

Setting up a CI pipeline

You are now ready to build a CI pipeline. As part of the demonstration in this section, you will build an **`nginx`** container with a small custom webpage loaded in it. After the container is built, you will push the **`nginx`** container to the container registry you created in the previous section. You will build the CI pipeline gradually over the next 13 steps:

1. To start, open the forked GitHub repo and open the folder for **`Chapter 15`**. In that folder, you will find a couple of files, including **`Dockerfile`** and **`index.html`**. These files are used to build the custom container. Throughout the example, you will make changes to **`index.html`** to trigger changes in the GitHub action. Let's have a look at the contents of **`index.html`**:

```
1  <html>
2  <head>
3      <title>Version 1</title>
4  </head>
5  <body>
6      <h1>Version 1</h1>
7  </body>
8  </html>
```

This is a simple HTML file, with a title and a header both saying **`Version 1`**. In the *Setting up a CD pipeline* section, you'll be asked to increment the version.

Next, you were also provided with a Dockerfile. The contents of that file are as follows:

```
1  FROM nginx:1.19.7-alpine
2  COPY index.html
/usr/share/nginx/html/index.html
```

This Dockerfile starts from an **`nginx-alpine`** base image. Nginx is a popular open-source web server, and Alpine is a lightweight operating system often used for container images. In the second line, you copy the local **`index.html`** file into the container, into the location where **`nginx`** loads webpages from.

Now that you have an understanding of the application itself, you're ready to start building the CI pipeline. For your reference, the full definition of the CI pipeline is provided as **`pipeline-ci.yaml`** in the code files with this chapter, but you'll be instructed to build this pipeline step by step in what follows.

2. Let's start by creating a GitHub Actions workflow. At the top of the screen in GitHub, click on Actions and then click on the set up a workflow yourself link, as shown in *Figure 15.11:*

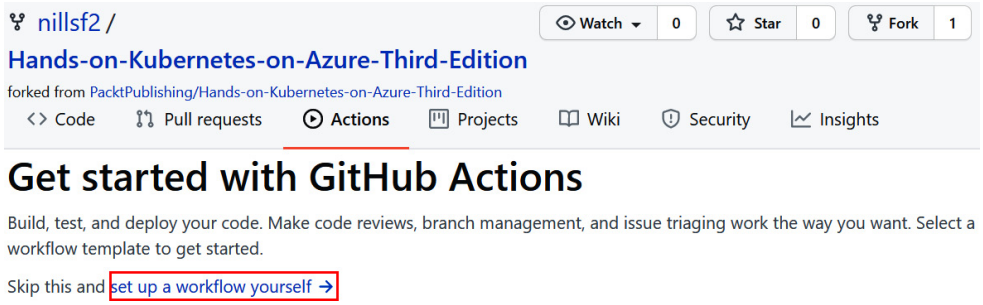

Figure 15.11: Creating a new GitHub action

3. This will take you to a code editor that is part of GitHub. First, change the name of the pipeline file to **`pipeline.yaml`** and change the name on *line 3* to **`pipeline`**, as shown in *Figure 15.12:*

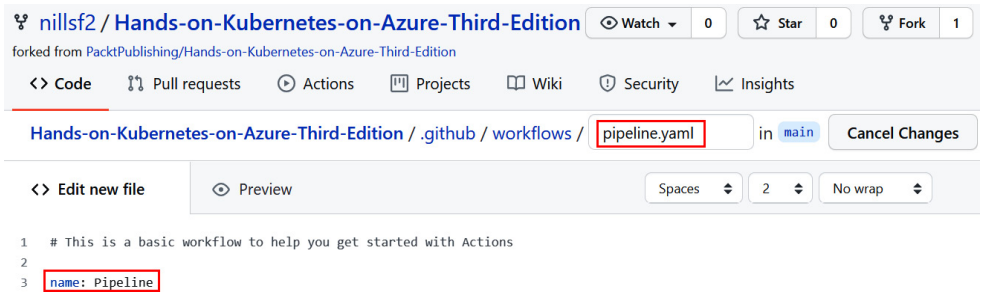

Figure 15.12: Changing the name of the pipeline

4. Next, you'll focus on the triggers of the workflow. In this demonstration, you'll only work with the main branch. However, you do not

want the workflow to run for every code change. You only want it to run when changes are made to either the pipeline definition or the code in the **`Chapter 15`** folder. To achieve this, you can set up the following code to control the workflow trigger:

```
4  # Controls when the action will run.
5  on:
6    # Triggers the workflow on push or pull request events but only for the main branch
7    push:
8      branches: [ main ]
9      paths:
10     - Chapter15/**
11     - .github/workflows/pipeline.yaml
12   # Allows you to run this workflow manually from the Actions tab
13   workflow_dispatch:
```

What this code configures is the following:

1. **Line 8**: Configures which branches will trigger this workflow. Specifically, in this case, this indicates that the workflow is triggered by pushing code to the main branch.
2. **Line 9-11**: This configures a path filter. Any changes in the **`Chapter15`** directory as well

as changes to the **`pipeline.yaml`** file in the **`.github/workflows/`** directory will trigger the workflow to run.

3. **Line 13**: This configures the workflow in such a way that it can be triggered manually as well. This means that you can trigger the workflow to run without making a code change.

You can also configure reusable variables in a GitHub Actions workflow. The following code block configures the container registry name you will use in multiple steps in the GitHub action:

```
14  # Env to set reusable variables
15  env:
16    ACRNAME: <acr-name>
```

Here, you are setting the **`ACRNAME`** variable to the name of the container registry you created. By using variables, you avoid having to configure the same value in multiple places.

That explains how the pipeline is triggered and how you can configure variables; let's now look at what will run in the pipeline.

5. Before we define the commands that are executed in the pipeline, let's explore the structure

of a GitHub Actions workflow, as shown in *Figure 15.13:*

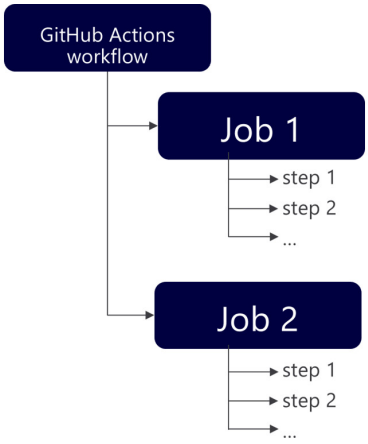

Figure 15.13: GitHub Actions workflow

A GitHub Actions workflow is made up of multiple jobs. A job can then have multiple steps in it. Jobs run in parallel by default but can be configured to run sequentially. The steps in a job will be run sequentially. A step in a job will contain the actual commands that will be run as part of the pipeline. An example of a step would be building a container image. There are multiple ways to run commands in a workflow: you can either run direct shell commands as you would on a regular terminal, or you can run prebuilt actions from the GitHub community.

The jobs and steps are run on what is called a runner. By default, workflows are run on hosted

runners. These hosted runners run on infrastructure set up and managed by GitHub. Optionally, you can run the jobs and steps on a self-hosted runner. This gives you the ability to have more configuration capabilities on the runner, for instance, to allow you to use special hardware or have specific software installed. Self-hosted runners can be physical, virtual, in a container, on-premises, or in a cloud.

In this section, you will run workflow steps from the community as well as shell commands. For an overview of actions available from the community, please refer to the GitHub marketplace at **https://github.com/marketplace?type=actions**.

In the CI pipeline you are building, you'll need to execute the following steps:

1. Get the GitHub repo on the action runner, also called a check-out of your repository.
2. Log in to the Azure CLI.
3. Log in to Azure Container Registry.
4. Build a container image and push this container image to Azure Container Registry.

Let's build the pipeline step by step.

1. Before you build the actual steps in the pipeline, you'll need to configure the jobs and the configuration of your job. Specifically, for this example, you can use the following configuration:

   ```
   18  jobs:
   19    # This workflow contains a single job called "CI"
   20    CI:
   21      # The type of runner that the job will run on
   22      runs-on: ubuntu-latest
   ```

   You are configuring the following:

   1. **Line 20**: You are creating a single job called **`CI`** for now. You'll add the CD job later.
   2. **Line 22**: This indicates that you'll run this job on a machine of type **`ubuntu-latest`**.

   This configures the GitHub runner for the steps. Let's now start building the individual steps.
2. The first step will be checking out the Git repo. This means that the code in the repo gets loaded by the runner. This can be achieved using the following lines of code:

```
25      steps:
26        # Checks-out your repository under
$GITHUB_WORKSPACE, so your job can access
it
27        - name: Git checkout
28          uses: actions/checkout@v2
```

The first line represented here (*line 25*) is what opens the **`steps`** block and all the following steps. The first step is called **`Git checkout`** (*line 27*) and simply refers to a prebuilt action called **`actions/checkout@v2`**. The **`@v2`** means that you are using the second version of this action.

3. Next, you will need to log in to the Azure CLI and then use the Azure CLI to log in to the Azure Container Registry. To do so, you'll make use of an action from the marketplace. You can find items in the marketplace by using the search bar at the right side of your screen, as shown in *Figure 15.14:*

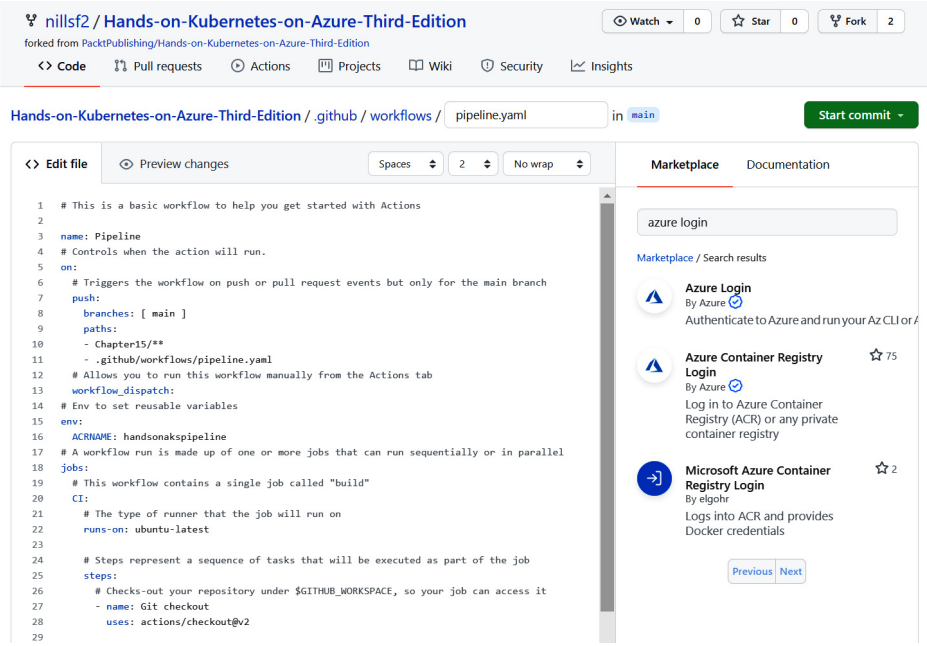

Figure 15.14: Searching for the Azure Login action

For this demonstration, you will use the Azure Login action. Click on the Azure Login action to get a screen with more information, as shown in *Figure 15.15:*

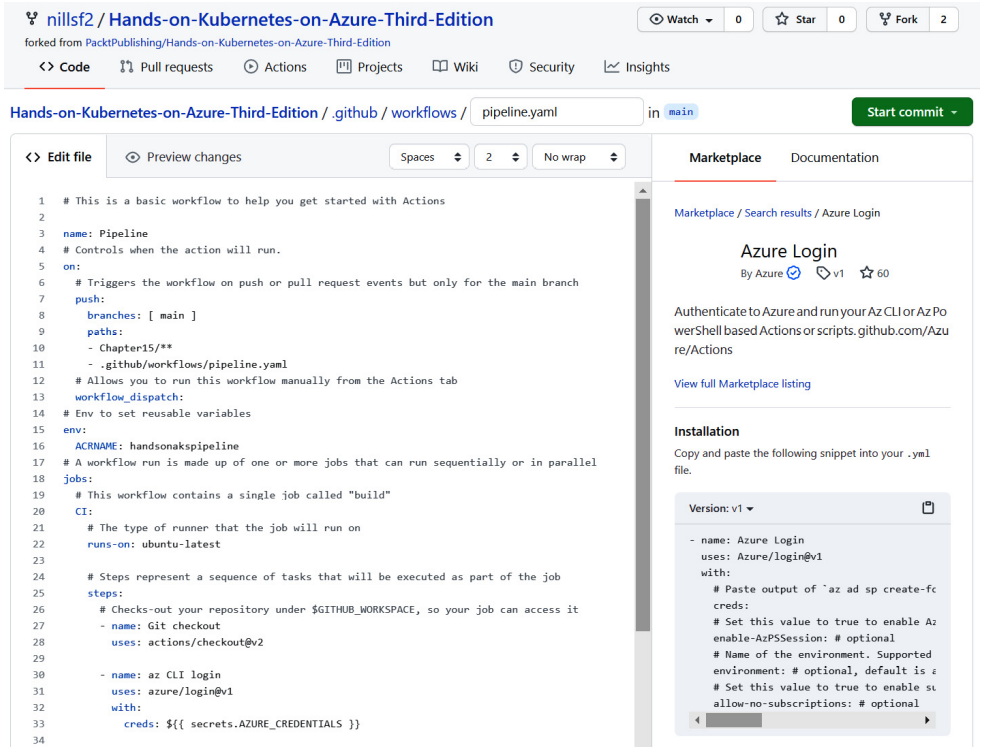

Figure 15.15: More details about the Azure Login action

This shows you more information on how to use that action and gives you sample code that you can copy and paste into the workflow editor.

To log in to the Azure CLI and Azure Container Registry, you can use the following code:

```
30      - name: az CLI login
31        uses: azure/login@v1
32        with:
33          creds: ${{
secrets.AZURE_CREDENTIALS }}
34
35      - name: ACR login
36        run: az acr login -n $ACRNAME
```

The first step logs in to the Azure CLI on the GitHub Actions runner. To log in to the Azure CLI, it uses the secret you configured in the previous section. The second job executes an Azure CLI command to log in to Azure Container Registry. It uses the variable you configured on *lines 14-15*. It executes the **`login`** command as a regular shell command. In the

next step, you'll push the image to this container registry.

4. Next, you build the container image. There are multiple ways to do this, and you'll use **`docker/build-push-action`** in this example:

```
39      - name: Build and push image
40        uses: docker/build-push-action@v2
41        with:
42          context: ./Chapter15
43          push: true
44          tags: ${{ env.ACRNAME }}.azurecr.io/website/website:${{ github.run_number }}
```

This step will build your container image and push it to the registry. You configure the context to run within the **`Chapter15`** folder, so the reference in the Dockerfile to the **`index.html`** page remains valid. It will tag that image with the name of your container registry, and as a version number for the container image, it will use the run number of the GitHub action. To get the run number of the workflow, you are using one of the default environment variables that GitHub configures. For a full list, please refer to the GitHub documentation:

https://docs.github.com/actions/reference/environment-variables.

Note

In this example, you are using the workflow run number as the version for your container image. Tagging container images is important since the tag version indicates the version of the container. There are multiple other strategies as well to version your container images.
One strategy that is discouraged is to tag container images with the latest tag and use that tag in your Kubernetes deployments. The `latest` *tag is the default tag that Docker will add to images if no tag is supplied. The problem with the* `latest` *tag is that if the image with the* `latest` *tag in your container registry changes, Kubernetes will not pick up this change directly. On nodes that have a local copy of the image with the* `latest` *tag, Kubernetes will not pull the new image until a timeout expires; however, nodes that don't have a copy of the image will pull the updated version when they need to run a pod with this image. This can cause you to have different versions running in a single deployment, which should be avoided.*

5. You are now ready to save and run this GitHub Actions workflow. You can save the workflow configuration file by clicking on the Start Commit button and then confirming by clicking Commit new file, as shown in *Figure 15.16:*

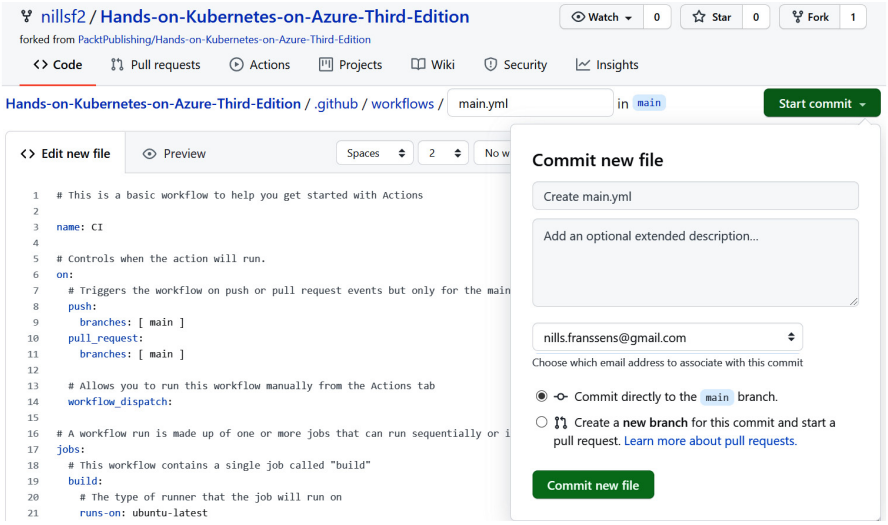

Figure 15.16: Saving the action configuration file

6. Once the file has been saved, the workflow will be triggered to run. To follow the logs of the workflow run, you can click on Actions at the top of the screen. This should show you a screen similar to *Figure 15.17:*

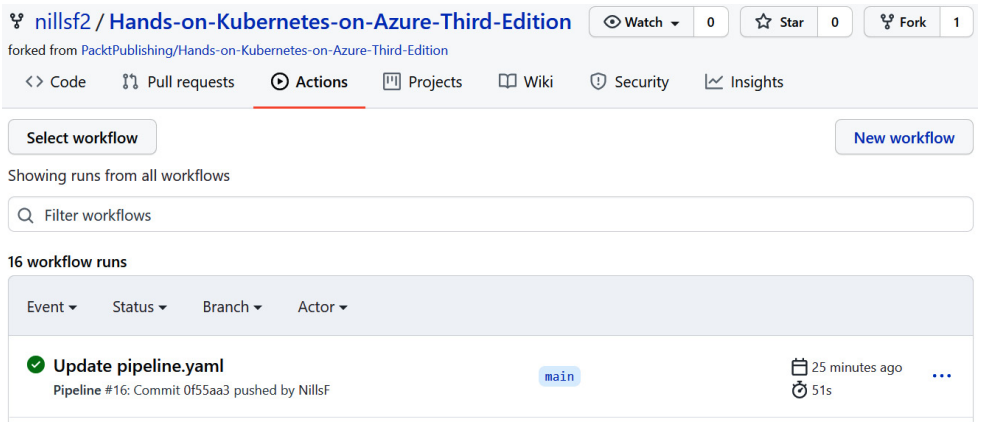

Figure 15.17: Getting the actions run history

Click on the top entry to get more details of your workflow run. This will bring you to a screen similar to *Figure 15.18:*

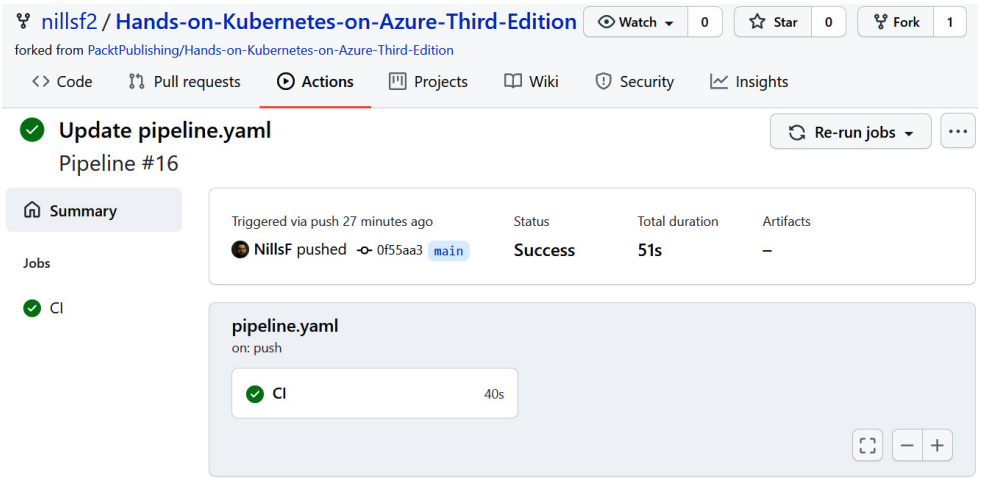

Figure 15.18: Detail screen of the action run

This shows you your workflow detail and shows you that you had a single job in your workflow. Click on CI to get the logs of that job. This will show you a screen similar to *Figure 15.19:*

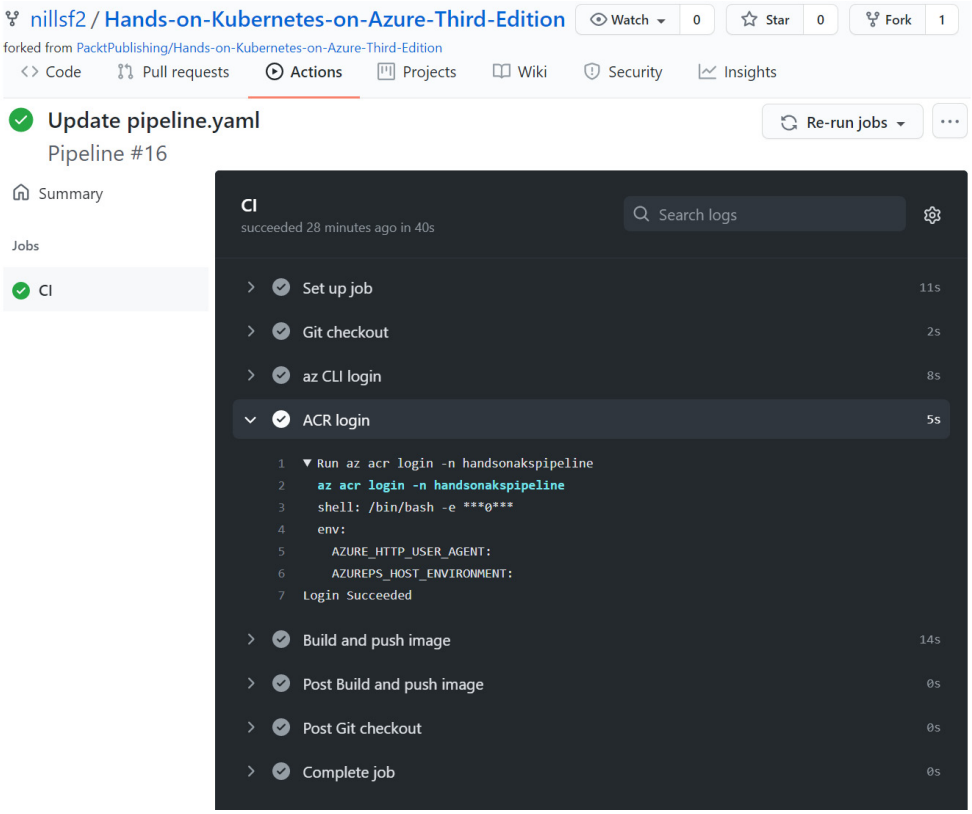

Figure 15.19: Logs of the CI job

On this screen, you can see the output logs of each step in your workflow. You can expand the logs of each step by clicking on the arrow icon in front of that step.

7. In this example, you built a container image and pushed that to a container registry on Azure. Let's verify this image was indeed pushed to the registry. For this, go back to the Azure portal and, in the search bar, look for **`container registry`**, as shown in *Figure 15.20:*

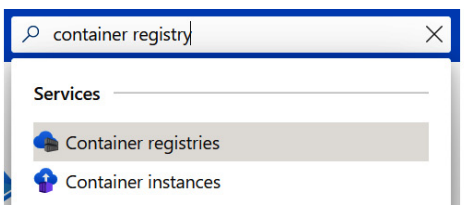

Figure 15.20: Navigating to the Container registries service through the Azure portal

In the resulting screen, click on the registry you created earlier. Now, click on Repositories on the left-hand side, which should show you the **`website/website`** repository, as shown in *Figure 15.21:*

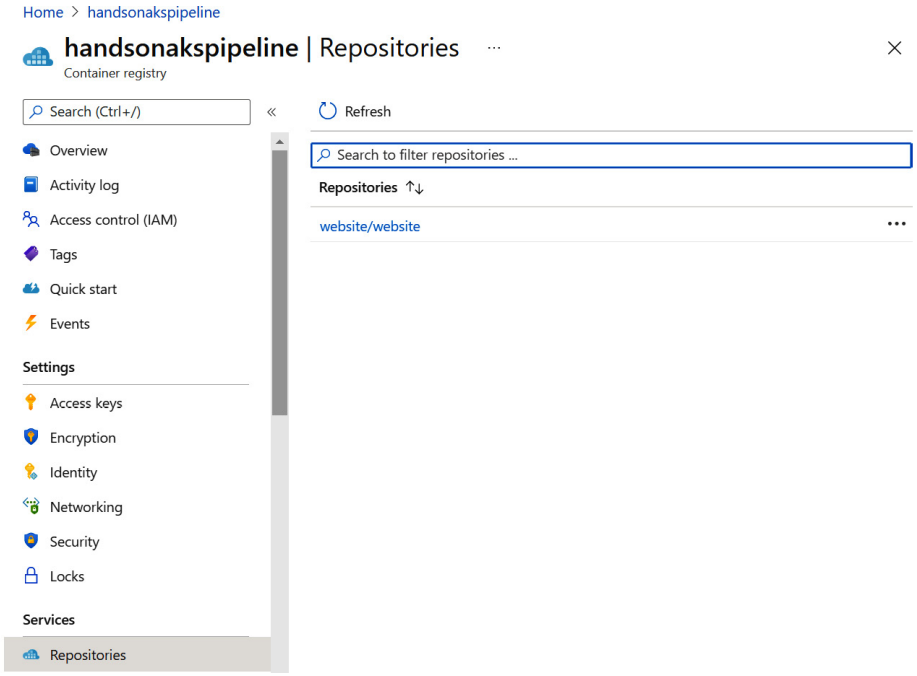

Figure 15.21: Showing the website/website repository in the container registry

8. If you click on the **`website/website`** repository link, you should see the image tags for your container image, as shown in *Figure 15.22:*

Figure 15.22: Image tags for the container image

If you compare the output of *Figure 15.18* and *Figure 15.22*, you will see that the run number of the action is also the tag on the image. In your case, that run number and tag will likely be 1.

You have now built a rudimentary CI pipeline. When the code in the **`Chapter 15`** folder is changed, the pipeline will run and build a new container image that will be pushed to the container registry. In the next section, you will add a CD job to your pipeline to also deploy the image to a deployment in Kubernetes.

Setting up a CD pipeline

You already have a pipeline with a CI job that will build a new container image. In this section, you'll add a CD job to that pipeline that will deploy the updated container image to a deployment in Kubernetes.

To simplify the application deployment, a Helm Chart for the application has been provided in the **`website`** folder inside **`Chapter 15`**. You can deploy the application by deploying the Helm Chart. By deploying using a Helm Chart, you can override the Helm values using the command line. You've done this in *Chapter 12, Connecting an app to an Azure database*, when you configured WordPress to use an external database.

In this CD job you will need to execute the following steps:

1. Check out the code.
2. Get AKS credentials.
3. Set up the application.
4. (Optional) Get the service's public IP.

Let's start building the CD pipeline. For your reference, the full CI and CD pipeline has been provided in the **`pipeline-cicd.yaml`** file:

1. To add the CD job to the pipeline, you'll need to edit the **`pipeline.yaml`** file. To do this, from within your forked repository, click on Code at the top of the screen and go to the **`.github/workflows`** folder. In that folder, click on the **`pipeline.yaml`** file. Once that file is open, click on the pen icon in the top right, as highlighted in *Figure 15.23:*

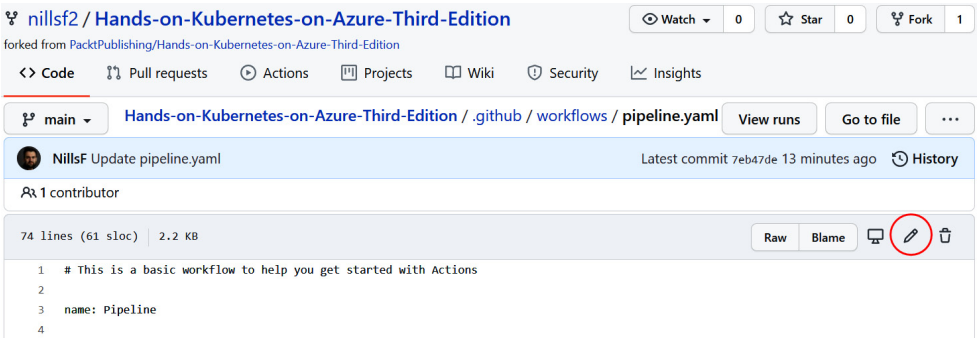

Figure 15.23: Editing the pipeline.yaml file

2. In the file, at the bottom, start by adding the following lines to define the CD job:

```
46  CD:
47    runs-on: ubuntu-latest
48    needs: CI
49    steps:
```

In this code block, you are creating the CD job. This will again run on an **`ubuntu-latest`** runner. On *line 48*, you are defining that this job has a dependency on the CI job. This means

that this job will only start after the CI job finishes, and it will only run if the CI job finishes successfully. Finally, *line 49* opens the **`steps`** block, which you will fill in next.

3. The first step will be a Git checkout. This will use the same step you use in the CI job as well:

```
50      - name: Git checkout
51        uses: actions/checkout@v2
```

4. Next, you'll need to log in to the Azure CLI and get the AKS credentials. You could do this by using the same approach as you did in the CI job, meaning you could do an Azure CLI login and then run the **`az aks get-credentials`** command on the runner. However, there is a single GitHub action that can achieve this for AKS:

```
53      - name: Azure Kubernetes set context
54        uses: Azure/aks-set-context@v1
55        with:
56          creds: ${{
secrets.AZURE_CREDENTIALS }}
57          resource-group: rg-handsonaks
58          cluster-name: handsonaks
```

This step uses the **`Azure/aks-set-context`** action from Microsoft. You configure it with the

Azure credentials secrets you created, and then define the resource group and cluster name you want to use. This will configure the GitHub action runner to use those AKS credentials.

5. You can now create the application on the cluster. As mentioned in the introduction of this section, you will deploy the application using the Helm Chart created in the **`website`** folder for this chapter. To deploy this Helm Chart on your cluster, you can use the following code:

```
60      - name: Helm upgrade
61        run: |
62          helm upgrade website
Chapter15/website --install \
63            --set
image.repository=$ACRNAME.azurecr.io/website/website
\
64            --set image.tag=${{
github.run_number }}
```

This code block executes a **`Helm upgrade`** command. The first argument (**`website`**) refers to the name of the Helm release. The second argument (**`Chapter15/website`**) refers to the location of the Helm Chart. The **`--install`** pa-

rameter configures Helm in such a way that if the chart isn't installed yet, it will be installed. This will be the case the first time you run this action.

In the following two lines, you set Helm values. You set the image repository to the **`website/website`** repo in your container registry, and you set the tag to the run number of the action. This is the same value you are using in the CI step to tag the image.

6. Finally, there is one optional step you can include in your workflow. This is getting the public IP address of the service that will be created to serve your website. This is optional because you could get this IP address using **`kubectl`** in Azure Cloud Shell, but it has been provided for your convenience:

```
66       - name: Get service IP
67         run: |
68           PUBLICIP=""
69           while [ -z $PUBLICIP ]; do
70             echo "Waiting for public IP..."
71             PUBLICIP=$(kubectl get service website -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

```
72          [ -z "$PUBLICIP" ] && sleep 10
73      done
74      echo $PUBLICIP
```

This code block will run a small Bash script. While the public IP hasn't been set, it will keep getting the public IP from the service using **`kubectl`**. Once the public IP has been set, the public IP will be shown in the GitHub Actions log.

7. You are now ready to save the updated pipeline and run it for the first time. To save the pipeline, click on the Start commit button at the top right of the screen and click on Commit changes in the pop-up window, as shown in *Figure 15.24:*

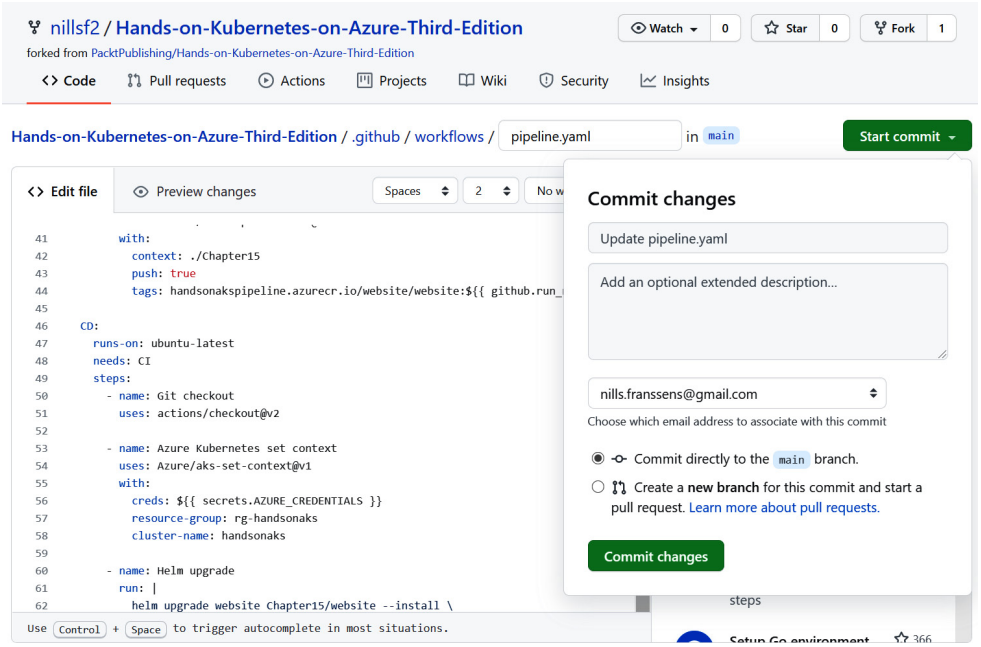

Figure 15.24: Pipeline workflow

8. Once you have committed the changes to GitHub, the workflow will be triggered to run. To follow the deployment, click on Actions at the top of the screen. Click on the top entry here to see the details of the run. Initially, the output will look similar to *Figure 15.25:*

Figure 15.25: Detailed output of the action run while the action is running

As you can see, you now have access to two jobs in this workflow run, the CI job and the CD job. While the CI job is running, the CD job's logs won't be available. Once the CI job finishes successfully, you'll be able to access the logs of the CD job. Wait for a couple of seconds until the screen looks like *Figure 15.26*, which indicates that the workflow successfully finished:

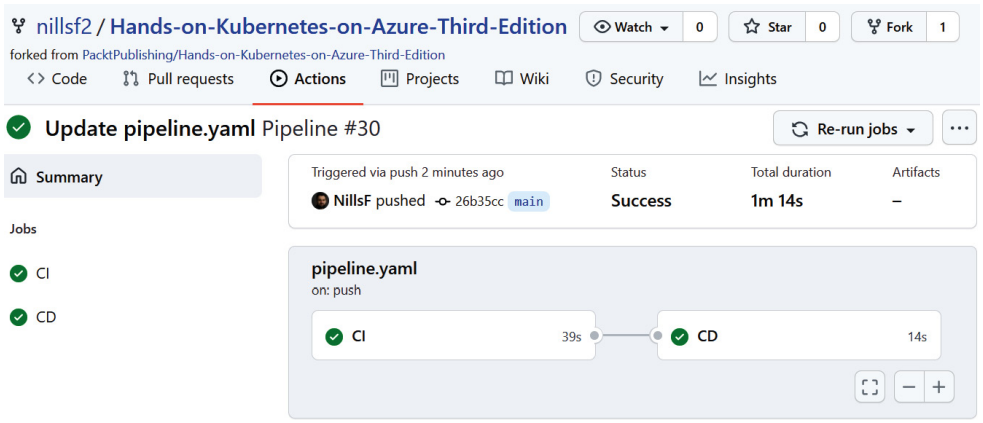

Figure 15.26: Detailed output of the action run after both jobs finished

9. Now, click on the CD job to see the logs of this job. Click on the arrow next to Get service IP to see the public IP of the service that got created, as shown in *Figure 15.27:*

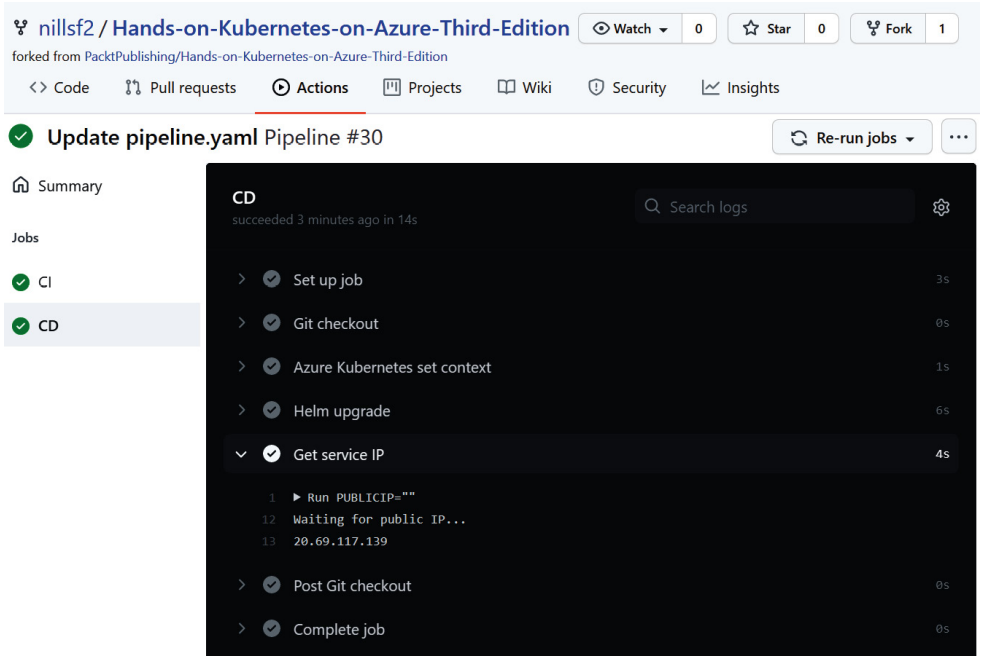

Figure 15.27: Logs of the CD job showing the public IP address of the service

Open a new tab in your web browser to visit your website. You should see an output similar to *Figure 15.28:*

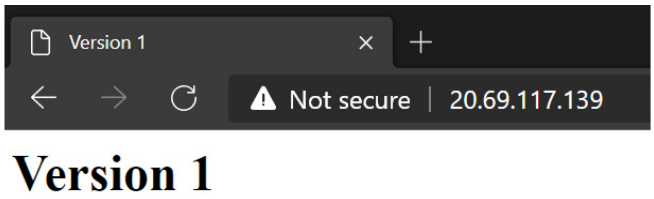

Figure 15.28: Website running version 1

10. Let's now test the end-to-end pipeline by making a change to the **`index.html`** file. To do this, in GitHub, click on Code at the top of the screen, open **`Chapter15`**, and click on the **`index.html`** file. In the resulting window, click on the pen icon in the top right, as shown in *Figure 15.29:*

Figure 15.29: Clicking on the pen icon to edit the index.html file

11. You can now edit the file. Change the version of the website to **`version 2`** (or any other changes you might want to make), and then scroll to the bottom of the screen to save the changes. Click on the Commit changes button to commit the changes to GitHub, as shown in *Figure 15.30:*

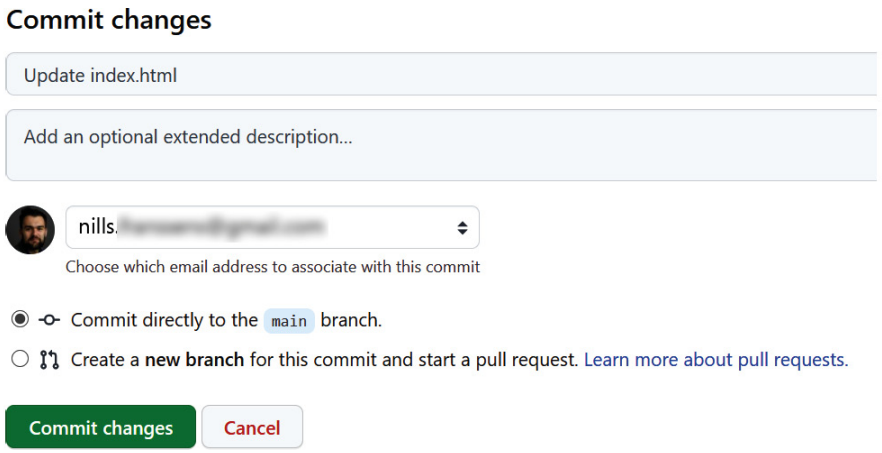

Figure 15.30: Saving the changes to the index.html file

12. This will trigger the workflow to be run. It will run through both the CI and CD jobs. This means that a new container will be built, with an updated **`index.html`** file. You can follow the status of the workflow run as you've done before, by clicking on Actions at the top of the screen and clicking on the top run. Wait until the job has finished, as shown in *Figure 15.31:*

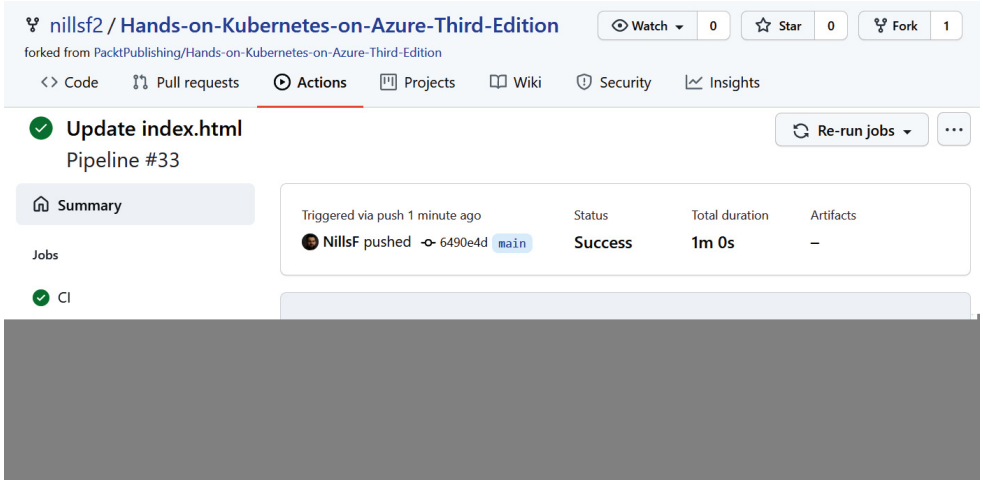

Figure 15.31: Action run after updating index.html

13. If you now browse back to the IP address you got as an output of *step 9*, you should see the updated webpage showing you Version 2, as shown in *Figure 15.32:*

Figure 15.32: The web page has been updated to version 2

This has shown you that the pipeline executed successfully and has brought your code changes to production.

Note

In this example, you updated the production version of your website directly, without any approvals. GitHub Actions also allows you to configure manual approvals in case you want to test changes before promoting them to production. To configure manual approvals, you can use the environments functionality in GitHub Actions. For more information, please refer to **https://docs.github.com/en/actions/reference/environments**

This concludes this example of CI and CD using GitHub Actions. Let's make sure to clean up the resource you created for this chapter. In Cloud Shell, execute the following commands:

```
helm uninstall website
```

```
az group delete -n rg-pipelines --yes
```

Since this also marks the end of the examples in this book, you can now also delete the cluster itself if you do not need it anymore. If you wish to do so, you can use the following command to delete the cluster:

```
az group delete -n rg-handsonaks --yes
```

This way, you ensure you aren't paying for the resources if you're no longer using them after you've finished the examples in this book.

Summary

You have now successfully created a CI/CD pipeline for your Kubernetes cluster. CI is the process of frequently building and testing software, whereas CD is the practice of regularly deploying software.

In this chapter, you used GitHub Actions as a platform to build a CI/CD pipeline. You started by building the CI pipeline. In that pipeline, you built a container image and pushed it to the container registry.

Finally, you also added a CD pipeline to deploy that container image to your Kubernetes cluster. You were able to verify that by making code changes to a webpage, the pipeline was triggered and code changes were pushed to your cluster.

The CI/CD pipeline you built in this chapter is a starter pipeline that lays the foundation for a more robust CI/CD pipeline that you can use to

deploy applications to production. You should consider adding more tests to the pipeline and also integrate it with different branches before using it in production.

Final thoughts

This chapter also concludes this book. During the course of this book, you've learned how to work with AKS through a series of hands-on examples.

The book started by covering the basics; you learned about containers and Kubernetes and you created an AKS cluster.

The next section focused on application deployment on AKS. You learned different ways of deploying applications to AKS, how to scale applications, how to debug failures, and how to secure services using HTTPS.

The next sections focused on security in AKS. You learned about role-based access control in Kubernetes and how you can integrate AKS with Azure Active Directory. Then, you learned about pod identities, and pod identities were used in a couple of follow-up chapters. After that, you

learned how to securely store secrets in AKS, and then we focused on network security.

The final section of this book focused on a number of advanced integrations of AKS with other services. You deployed an Azure database through the Kubernetes API and integrated it with a WordPress application on your cluster. You then explored how to monitor configuration and remediate threats on your cluster using Azure Security Center. You then deployed Azure functions on your cluster and scaled them using KEDA. In this final chapter, you configured a CI/CD pipeline to automatically deploy an application to Kubernetes based on code changes.

If you've successfully completed all the examples provided in this book, you should now be ready to build and run applications at scale on top of AKS.