10 Managing Containers Effectively with Kubernetes

In the previous chapter, we learned in detail about containers with Docker, about the construction of a Docker image, and about the instantiation of a new container on the local machine. Finally, we set up a **continuous integration/continuous deployment** (**CI/CD**) pipeline that builds an image, deploys it in Docker Hub, and executes its container in **Azure Container Instances** (**ACI**).

All this works well and does not pose too many problems when working with a few containers. But in so-called **microservice applications**—that is, applications that are composed of several services (each of them is a container), we will need to manage and orchestrate these containers.

There are two major container orchestration tools on the market: Docker Swarm and Kubernetes.

For some time now, Kubernetes, also known as **K8S**, has proved to be a true leader in the field of container management and is therefore becoming a *must* for the containerization of applications.

In this chapter, we will learn how to install Kubernetes on a local machine, as well as an example of how to deploy an application in Kubernetes, both in a standard way and with Helm. We will learn in more depth about Helm by creating a chart and publishing it in a private registry on **Azure Container Registry** (**ACR**).

Then, we will talk about **Azure Kubernetes Service** (**AKS**) as an example of a Kubernetes cluster, and finally, we will learn how to monitor applications and metrics in Kubernetes.

This chapter will cover the following topics:

- Installing Kubernetes
- A first example of Kubernetes application deployment
- Using Helm as a package manager
- Publishing a Helm chart in a private registry (ACR)
- Using AKS
- Creating a CI/CD pipeline for Kubernetes with Azure Pipelines
- Monitoring applications and metrics in Kubernetes

Technical requirements

This chapter is a continuation of the previous chapter on Docker, so to understand it properly, it is necessary to have read that chapter and to have installed Docker Desktop (for the Windows **operating system** (**OS**)).

In the CI/CD part of this chapter, you will need to retrieve the source code that was provided in the previous chapter on Docker, which is available at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09/appdocker`.

For the section about Helm charts on ACR, it's necessary to have an Azure subscription (register for free here: `https://azure.microsoft.com/en-us/free/`) and that you have installed the Azure **Command-Line Interface** (**CLI**) binary, available here: `https://docs.microsoft.com/en-us/cli/azure/install-azure-cli`.

The entire source code for this chapter is available at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP10`.

Check out the following video to see the Code in Action:

`https://bit.ly/3p7ydjt`

Installing Kubernetes

Before installing Kubernetes, we need to have an overview of its architecture and main components, because Kubernetes is not a simple tool but is a cluster—that is, it consists of a **master** server and other slave servers called **nodes**.

I suggest you explore the architecture of Kubernetes in a simplified way.

Kubernetes architecture overview

Kubernetes is a platform that is made up of several components that assemble together and extend on demand, in order to enable better scalability of applications. The architecture of Kubernetes, which is a client/server type, can be represented simply, as shown in the following diagram:

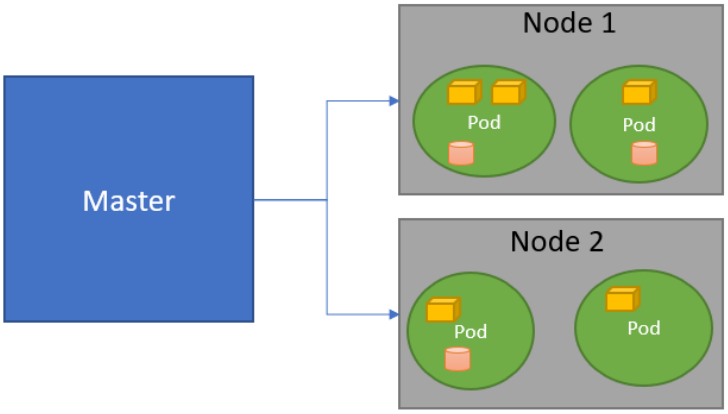


Figure 10.1 – Kubernetes architecture

In the previous diagram, we can see that a cluster is made up of a master component and nodes (also called **worker** nodes), which represent the slave servers.

In each of these nodes, there are **pods**, which are virtual elements that will contain containers and volumes.

Put simply, we can create one pod per application, and it will contain all the containers of the application. For example, one pod can contain a web server container, a database container, and a volume that will contain persistent files for images and database files.

Finally, `kubectl` is the client tool that allows us to interact with a Kubernetes cluster. With this, we have the main requirements that allow us to work with Kubernetes, so let's look at how we can install it on a local machine.

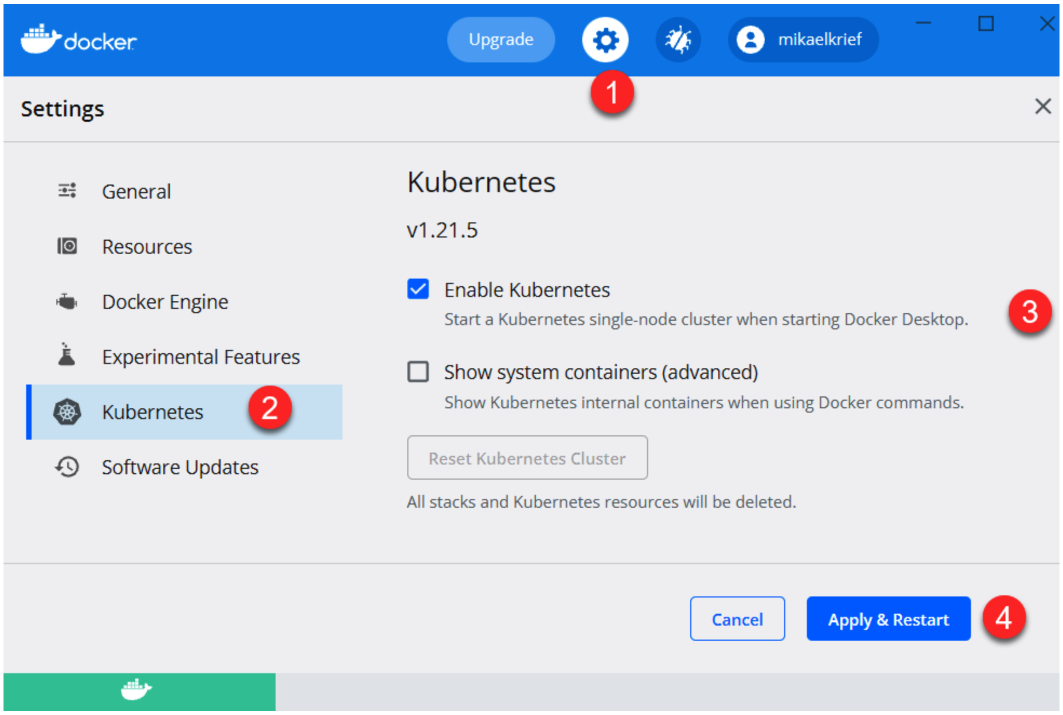
Installing Kubernetes on a local machine

When developing a containerized application that is to be hosted on Kubernetes, it is very important to be able to run the application (with its containers) on your local machine, before deploying it on remote Kubernetes production clusters.

In order to install a Kubernetes cluster locally, there are several solutions, which are detailed next.

The first solution is to use **Docker Desktop** by performing the following steps:

1. If we have already installed Docker Desktop, which we learned about in *Chapter 9*, *Containerizing Your Application with Docker*, we can activate the **Enable Kubernetes** option in **Settings** on the **Kubernetes** tab, as shown in the following screenshot:



Kubernetes

v1.21.5

☑ Enable Kubernetes
Start a Kubernetes single-node cluster when starting Docker Desktop.

☐ Show system containers (advanced)
Show Kubernetes internal containers when using Docker commands.

All stacks and Kubernetes resources will be deleted.

Figure 10.2 – Enabling Kubernetes in Docker Desktop

2. After clicking on the **Apply & Restart** button, Docker Desktop will install a local Kubernetes cluster, and also the `kubectl` client tool, on the local machine.

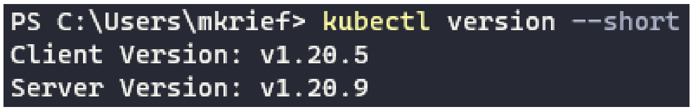
The second way of installing Kubernetes locally is to install `minikube`, which also installs a simplified Kubernetes cluster locally. Here is the official documentation that you can read: `https://minikube.sigs.k8s.io/docs/start/`.

> **Note**
>
> There are other solutions for installing local Kubernetes, such as `kind` or `kubeadm`. For more details, read the documentation here: `https://kubernetes.io/docs/tasks/tools/`.

Following the local installation of Kubernetes, we will check its installation by executing the following command in a Terminal:

```
kubectl version --short
```

The following screenshot shows the results for the preceding command:



```
PS C:\Users\mkrief> kubectl version --short
Client Version: v1.20.5
Server Version: v1.20.9
```

Figure 10.3 – kubectl getting the binary version

> **Note**
>
> All of the operations that we carry out on our Kubernetes cluster will be done with `kubectl` commands.

After installing our Kubernetes cluster, we'll need another element, which is the Kubernetes dashboard. This is a web application that allows us to view the status, as well as all the components, of our cluster.

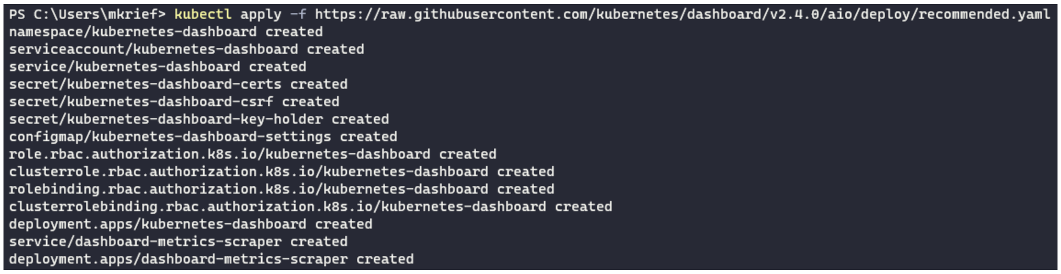
In the next section, we'll discuss how to install and test the Kubernetes dashboard.

Installing the Kubernetes dashboard

In order to install the Kubernetes dashboard, which is a pre-packaged containerized web application that will be deployed in our cluster, we will run the following command in a Terminal:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
dashboard/v2.4.0/aio/deploy/recommended.yaml
```

Its execution is shown in the following screenshot:



```
PS C:\Users\mkrief> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

Figure 10.4 – Kubernetes dashboard installation

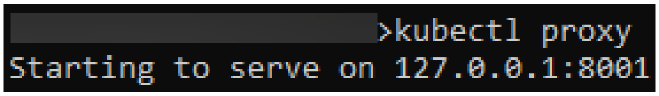
From the preceding screenshot, we can see that different artifacts are created, which are outlined as follows: secrets, two web applications, **role-based access control** (**RBAC**) roles, permissions, and services.

> **Note**
>
> Note that the **Uniform Resource Locator** (**URL**) mentioned in the parameters of the command that installs the dashboard may change depending on the versions of the dashboard. To find out the last valid URL to date, consult the official documentation by visiting `https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/`.

Now that we have installed this Kubernetes dashboard, we will connect to it and configure it.

To open the dashboard and connect to it from our local machine, we must first create a proxy between the Kubernetes cluster and our machine by performing the following steps:

1. To create a proxy, we execute the `kubectl proxy` command in a Terminal. The detail of the execution is shown in the following screenshot:



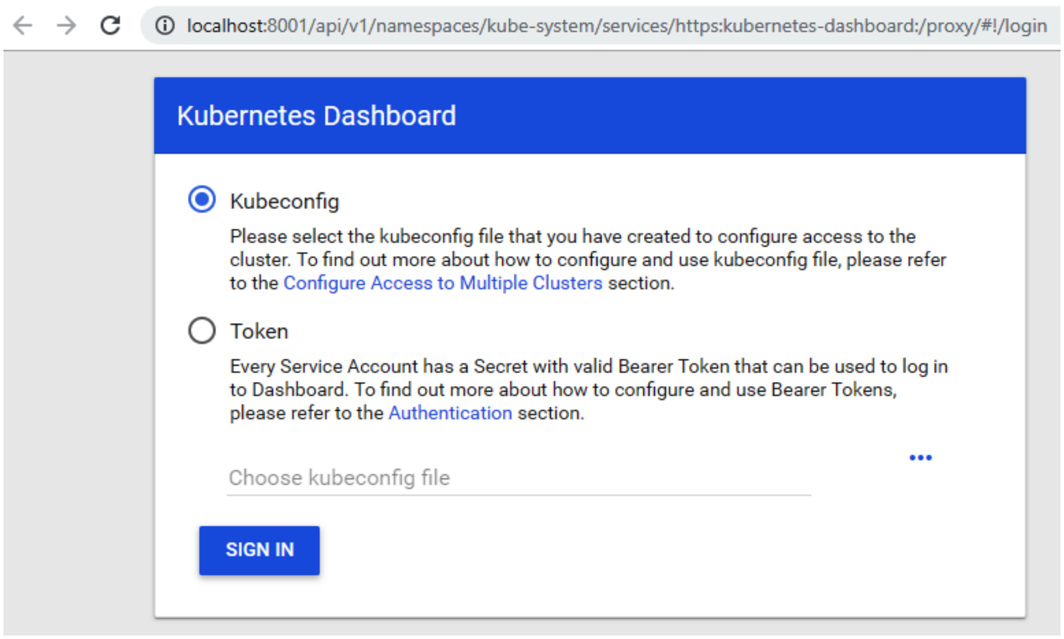
```
[redacted]>kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Figure 10.5 – kubectl proxy command

   We can see that the proxy is open on the localhost address (`127.0.0.1`) on port `8001`.

2. Then, in a web browser, open the following URL, `http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/login`, which is a local URL (localhost and `8001`) that is created by the proxy and that points to the Kubernetes dashboard application we have installed.

   The following screenshot shows how to select the Kubernetes configuration file or enter the authentication token:



localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/login

Kubernetes Dashboard

[✓ Kubeconfig]

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

Choose kubeconfig file

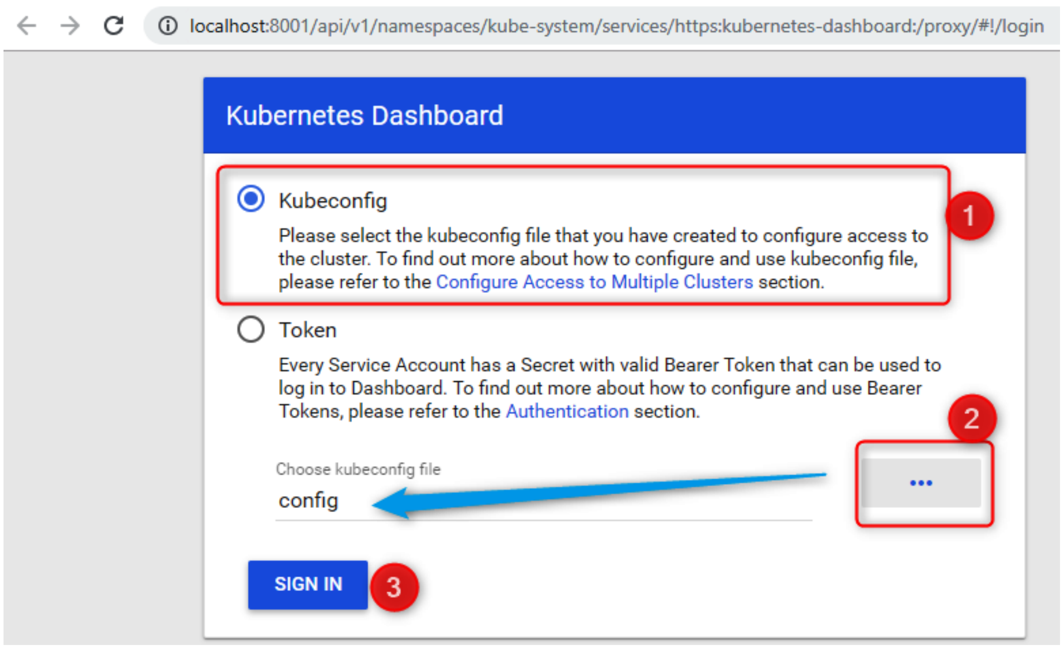
Figure 10.6 – Kubernetes dashboard authentication

3. To create a new user authentication token, we will execute the following script in a PowerShell Terminal:

```
$TOKEN=((kubectl -n kube-system describe secret default |
Select-String "token:") -split " +")[1]
kubectl config set-credentials docker-for-desktop
--token="${TOKEN}"
```

The execution of this script creates a new token inside the local config file.

4. Finally, in the dashboard, we will select the `config` file, which is located in the `C:\Users\<user name>.kube\` folder, as shown in the following screenshot:



localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/login

Kubernetes Dashboard

[● Kubeconfig]

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

○ Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

Choose kubeconfig file: config

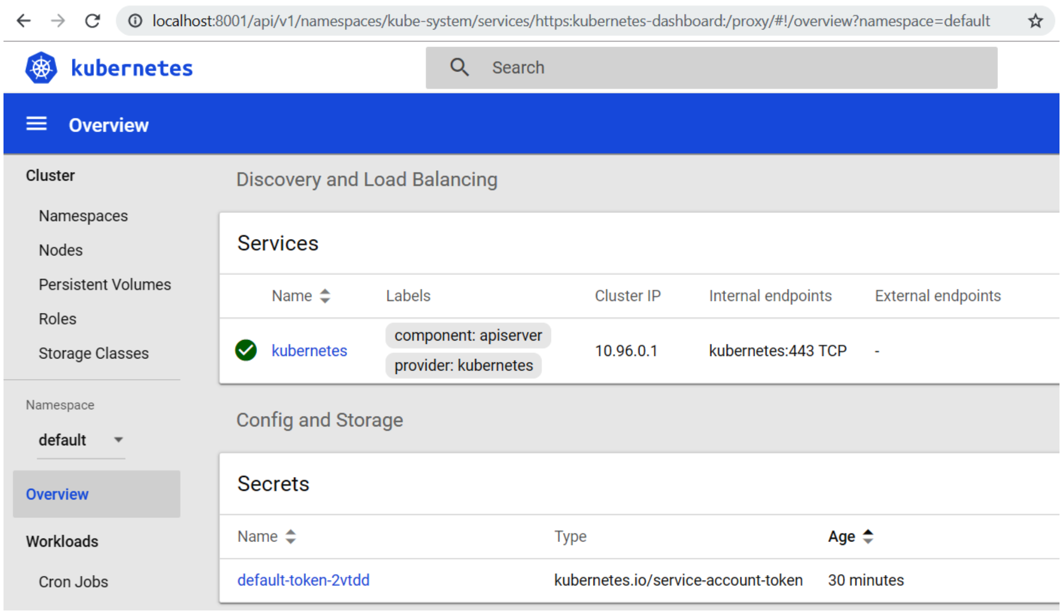
Figure 10.7 – Kubernetes dashboard authentication with the kubeconfig file

Note

For token authentication, read this blog post:

`https://www.replex.io/blog/how-to-install-access-and-add-heapster-metrics-to-the-kubernetes-dashboard`

5. After clicking on the **SIGN IN** button, the dashboard is displayed, as follows:



localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/overview?namespace=default

Name	Labels	Cluster IP	Internal endpoints	External endpoints
✓ kubernetes	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP	-

Name	Type	Age
default-token-2vtdd	kubernetes.io/service-account-token	30 minutes

Figure 10.8 – Kubernetes dashboard resources list

We have just seen how to install a Kubernetes cluster on a local machine, and then we installed and configured the Kubernetes web dashboard in this cluster. We will now deploy our first application in the local Kubernetes cluster using **YAML Ain't Markup Language** (**YAML**) specification files and `kubectl` commands.

A first example of Kubernetes application deployment

After installing our Kubernetes cluster, we will deploy an application in it. First of all, it is important to know that when we deploy an application in Kubernetes, we create a new instance of the Docker container in a Kubernetes **pod** object, so we first need to have a Docker image that contains the application.

For our example, we will use the Docker image that contains a web application that we have pushed into Docker Hub in *Chapter 9*, *Containerizing Your Application with Docker*.

To deploy this instance of the Docker container, we will create a new `k8sdeploy` folder, and inside it, we will create a Kubernetes deployment YAML specification file (`myapp-deployment.yml`) with the following content:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
     name: webapp
spec:
     selector:
     matchLabels:
          app: webapp
 replicas: 2
 template:
    metadata:
     labels:
       app: webapp
   spec:
      containers:
      - name: demobookk8s
        image: mikaelkrief/demobook:latest
        ports:
        - containerPort: 80
```

In this preceding code snippet, we describe our deployment in the following way:

- The `apiVersion` property is the version of `api` that should be used.
- In the `Kind` property, we indicate that the specification type is deployment.
- The `replicas` property indicates the number of pods that Kubernetes will create in the cluster; here, we choose two instances.

In this example, we chose two replicas, which can—at the very least—distribute the traffic load of the application (if there is a high volume of load, we can put in more replicas), while also ensuring the proper functioning of the application. Therefore, if one of the two pods has a problem, the other (which is an identical replica) will ensure the proper functioning of the application.

Then, in the `containers` section, we indicate the image (from Docker Hub) with `name` and `tag`. Finally, the `ports` property indicates the port that the container will use within the cluster.

> **Note**
>
> This source code is also available at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP10/k8sdeploy/myapp-deployment.yml`.

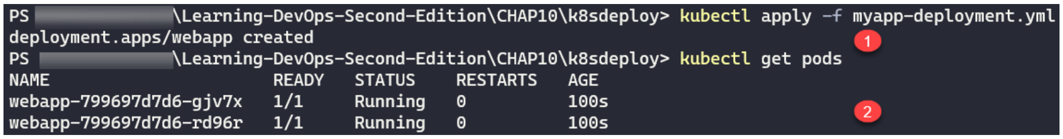
To deploy our application, we go to our Terminal and execute one of the essential `kubectl` commands (`kubectl apply`), as follows:

```
kubectl apply -f myapp-deployment.yml
```

The `-f` parameter corresponds to the YAML specification file.

This command applies the deployment that is described in the YAML specification file on the Kubernetes cluster.

Following the execution of this command, we will check the status of this deployment by displaying a list of pods in the cluster. To do this in the Terminal, we execute the `kubectl get pods` command, which returns a list of cluster pods. The following screenshot shows the execution of the deployment and displays the information in the pods, which we use to check the deployment:



```
PS [redacted]\Learning-DevOps-Second-Edition\CHAP10\k8sdeploy> kubectl apply -f myapp-deployment.yml
deployment.apps/webapp created
PS [redacted]\Learning-DevOps-Second-Edition\CHAP10\k8sdeploy> kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
webapp-799697d7d6-gjv7x  1/1     Running   0          100s
webapp-799697d7d6-rd96r  1/1     Running   0          100s
```

Figure 10.9 – kubectl apply command

What we can see in the preceding screenshot is that the second command displays our two pods, with the name (`webapp`) specified in the YAML file, followed by a **unique identifier** (**UID**), and that they are in a `Running` status.

We can also visualize the status of our cluster on the Kubernetes web dashboard, the `webapp` deployment with the Docker image that has been used, and the two pods that have been created. For more details, we can click on the different links of the elements.

Our application has been successfully deployed in our Kubernetes cluster but, for the moment, it is only accessible inside the cluster, and for it to be usable, we need to expose it outside the cluster.

In order to access the web application outside the cluster, we must add a service type and a `NodePort` category element to our cluster. To add this service type and `NodePort` element, in the same way as for deployment, we will create a second YAML file (`myapp-service.yml`) of the service specification in the same `k8sdeploy` directory, which has the following code:

```
---
apiVersion: v1
kind: Service
metadata:
 name: webapp
 labels:
 app: webapp
spec:
 type: NodePort
 ports:
 - port: 80
   targetPort: 80
   nodePort: 31000
selector:
 app: webapp
```

In the preceding code snippet, we specify the kind, `Service`, as well as the type of service, `NodePort`.

Then, in the `ports` section, we specify the port translation: port `80`, which is exposed internally, and port `31000`, which is exposed externally to the cluster.

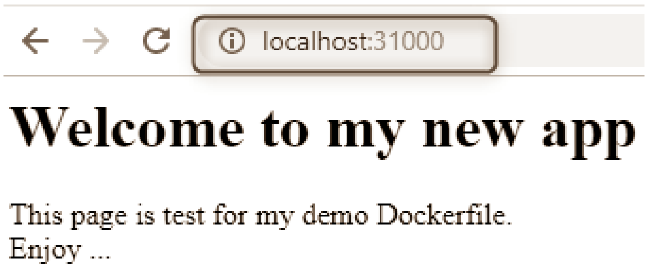
Note

The source code of this file is also available at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP10/k8sdeploy/myapp-service.yml`.

To create this service on the cluster, we execute the `kubectl apply` command, but this time with our `myapp-service.yaml` file as a parameter, as follows:

```
kubectl apply -f myapp-service.yml
```

The execution of the command creates the service within the cluster, and, to test our application, we open a web browser with the `http://localhost:31000` URL, and our page is displayed as follows:



Welcome to my new app

This page is test for my demo Dockerfile.
Enjoy ...

Figure 10.10 – Demo Kubernetes application

Our application is now deployed on a Kubernetes cluster, and it can be accessed from outside the cluster.

In this section, we have learned that the deployment of an application, as well as the creation of objects in Kubernetes, is achieved using specification files in YAML format and several `kubectl` command lines.

The next step is to use Helm packages to simplify the management of the YAML specification files.

Using Helm as a package manager

As previously discussed, all the actions that we carry out on the Kubernetes cluster are done via the `kubectl` tool and the **YAML specification files**.

In a company that deploys several microservice applications on a Kubernetes cluster, we often notice a large number of these YAML specification files, and this poses a maintenance problem. In order to solve this maintenance problem, we can use **Helm**, which is the package manager for Kubernetes.

> **Note**
> For more information on package managers, you can also read the *Using a package manager* section of *Chapter 7, Continuous Integration and Continuous Delivery*.

Helm is, therefore, a repository that will allow the sharing of packages called **charts** that contain ready-to-use Kubernetes specification file templates.

11
Testing APIs with Postman

In the previous chapters, we talked about DevOps culture and **Infrastructure as Code** (**IaC**) with Terraform, Ansible, and Packer. Then, we saw how to use a source code manager with Git, along with the implementation of a CI/CD pipeline with Jenkins and Azure Pipelines. Finally, we showed the containerization of applications with Docker and their deployment in a Kubernetes cluster.

If you are a developer, you should realize that you use APIs every day, either for client-side use (where you consume the API) or as a provider of the API.

An API, as well as an application, must be testable, that is, it must be possible to test the different methods of this API in order to verify that it responds without error and that the response of the API is equal to the expected result.

In addition, the proper functioning of an API is much more critical to an application because this API is potentially consumed by several client applications, and if it does not work, it will have an impact on all of these applications.

The common API challenges are that we need to script or develop a dedicated application client to test each API individually or a workflow of multiple API execution.

In this chapter, we will learn how to test an API with a specialized tool called **Postman**. We will explore the use of collections and variables, then we will write Postman tests, and finally, we will see how to automate the execution of Postman tests with Newman in a CI/CD pipeline.

This chapter covers the following topics:

- Creating a Postman collection
- Using environments and variables
- Writing Postman tests
- Executing tests locally
- Understanding the Newman concept
- Preparing Postman collections for Newman
- Running the Newman command line
- Integration of Newman in the CI/CD pipeline process

Technical requirements

In this chapter, we will use **Newman**, which is a Node.js package. Therefore, we need to install Node.js and npm on our computer beforehand, which we can download at `https://nodejs.org/en/`.

For the demo APIs that are used in this chapter, we will use an example that is provided on the internet: `https://jsonplaceholder.typicode.com/`.

The GitHub repository, which contains the complete source code used in this chapter, can be found at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP11`.

Check out the following video to see the code in action: `https://bit.ly/3s7239U`.

Creating a Postman collection with requests

Postman is a free client tool in a graphical format that can be installed on any type of OS. Its role is to test APIs through requests, which we will organize into collections. It also allows us to dynamize API tests through the use of variables and the implementation of environments. Postman is famous for its ease of use, but also for the advanced features that it offers.

In this section, we will learn how to create and install a Postman account, then we will create a collection that will serve as a folder to organize our requests, and finally, we will create a request that will test a demo API.

Before we use Postman, we will need to create a Postman account by going to `https://www.postman.com/` and clicking on the **Sign Up for Free** button. In the form, click on the **Create Account** link, as shown in the following screenshot:

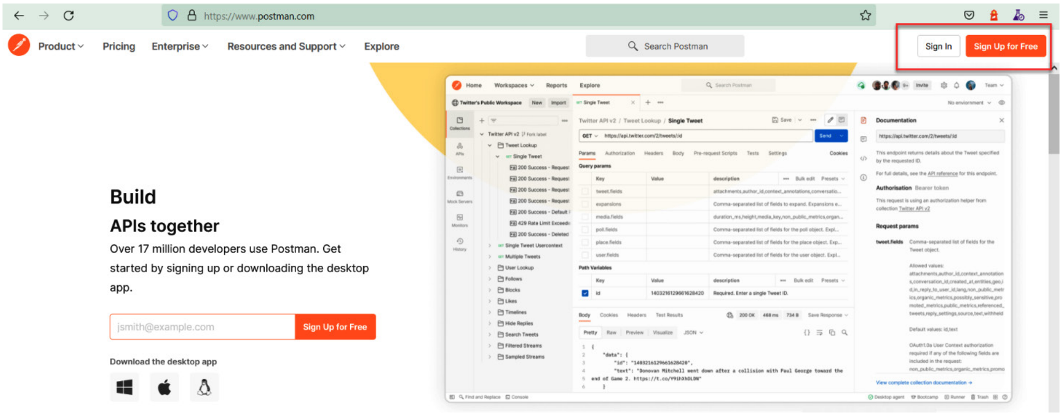


Figure 11.1 – Postman signup

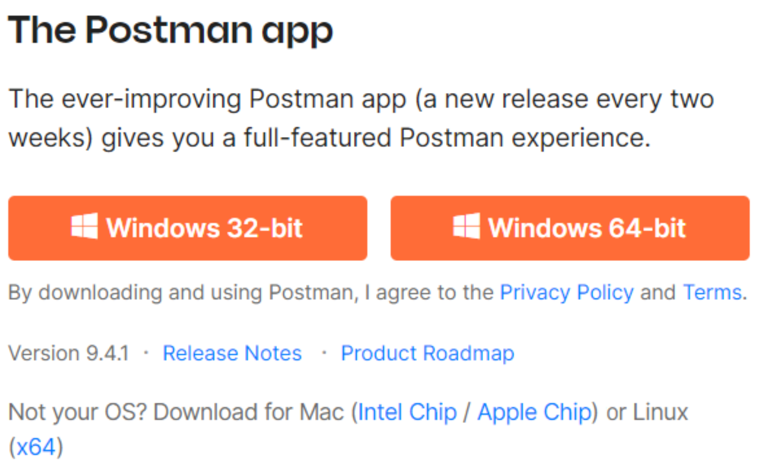
Now, you can either create a Postman account for yourself by filling out the form, or you can create an account using your Google account.

This account will be used to synchronize Postman data between your machine and your Postman account. This way, the data will be accessible on all of your workstations.

After creating a Postman account, we will look at how to download and install it on a local machine.

Installation of Postman

Once the Postman account has been created, those who are using Windows can download Postman from `https://www.getpostman.com/downloads/` and choose the version to install. For those who want to install it on Linux or macOS, just click on the link of your OS. The following screenshot shows the download links according to your OS:



The Postman app

The ever-improving Postman app (a new release every two weeks) gives you a full-featured Postman experience.

By downloading and using Postman, I agree to the Privacy Policy and Terms.

Version 9.4.1 · Release Notes · Product Roadmap

Not your OS? Download for Mac (Intel Chip / Apple Chip) or Linux (x64)

Figure 11.2 – Postman download

Once Postman is downloaded, we need to install it by clicking on the download file for Windows, or for other OSes, follow the installation documentation at `https://learning.getpostman.com/docs/postman/launching_postman/installation_and_updates/`.

We have just seen that the installation of Postman is very simple; the next step is to create a collection in which we will create a request.

Note

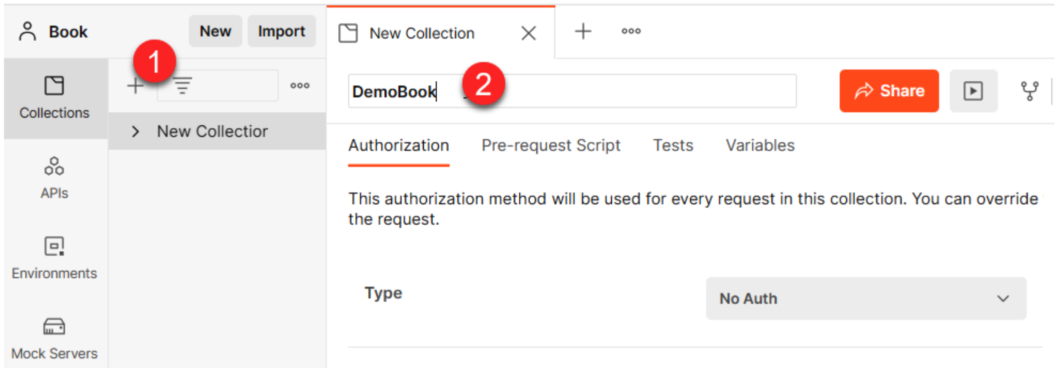
The API that we will test in this chapter is a demo API, which is provided freely on the following site: `https://jsonplaceholder.typicode.com/`.

Creating a collection

In Postman, any request that we test must be added to a directory called `Collection`, which provides storage for requests and allows for better organization.

We will, therefore, create a `DemoBook` collection that will contain the requests to the demo API, and for this, we will perform the following tasks:

1. In Postman, in the left-hand panel, click on the **Collections** | + button.
2. Once the tab opens, we will enter the name `DemoBook`. These steps for creating a new collection are illustrated in the following screenshot:



This authorization method will be used for every request in this collection. You can override the request.

Type: No Auth

Figure 11.3 – Postman collection creation

So, we have a `Demobook` collection that appears in the left-hand panel of Postman.

This collection is also synchronized with our Postman web account, and we can access it at `https://web.postman.co/me/collections`.

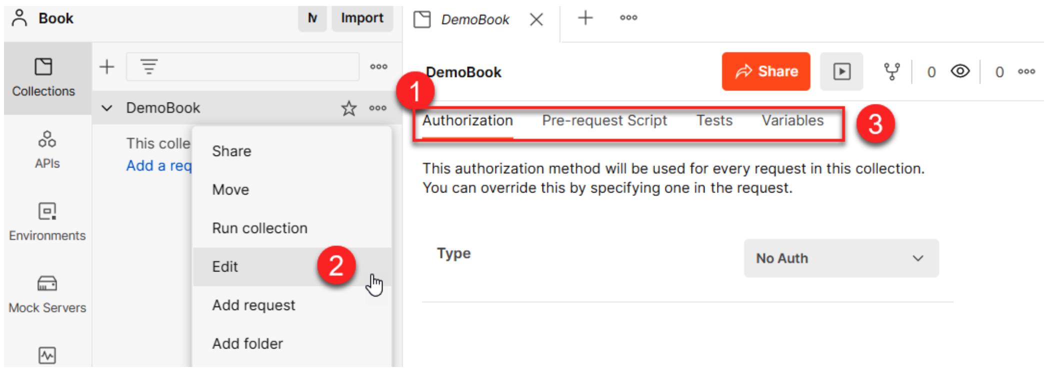
This collection will allow us to organize the requests of our API tests, and it is also possible to modify its properties in order to apply a certain configuration to all the requests that will be included in this collection.

These properties include request authentication, tests to be performed before and after requests, and variables common to all requests in this collection.

To modify the settings and properties of this collection, perform the following actions:

1. Click on the **...** button of the context menu of the collection.
2. Choose the **Edit** option, and the edit form appears, in which we can change all the settings that will apply to the requests in this collection.
3. Switch between all configuration tabs for the edit authorization, scripts, tests, or variables options.

The following screenshot shows the steps that are taken to modify the properties of a collection:



This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

Figure 11.4 – Postman edit collection

So, we have discussed the procedure that is followed in order to create a collection that is the first Postman artifact, and this will allow us to organize our API test requests.

We will now create a request that will call and test the proper functioning of our demo API.

Creating our first request

In Postman, the object that contains the properties of the API to be tested is called a **request**.

This request contains the configuration of the API itself, but it also contains the tests that are to be performed to check that it is functioning properly.

The main parameters of a request are as follows:

- The URL of the API
- Its method: `GET/POST/DELETE/PATCH`
- Its authentication properties
- Its querystring keys and its body request
- The tests that are to be performed before or after execution of the API

The creation of a request is done in two steps – its creation in the collection, followed by its configuration.

1. **The creation of the request**: To create the request of our API, here are the steps that need to be followed:

   I. We go to the context menu of the `DemoBook` collection and click on the **Add Request** option:

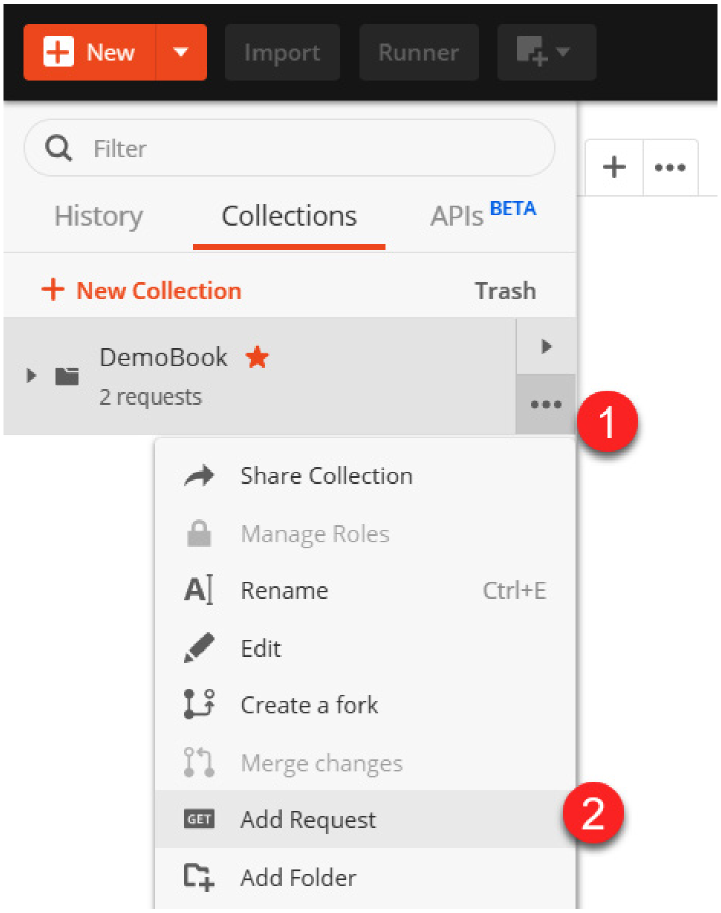


Figure 11.5 – Add Request in Postman

II. Then, in the new tag, enter the name of the request, `Get all posts`, as shown in the following screenshot:

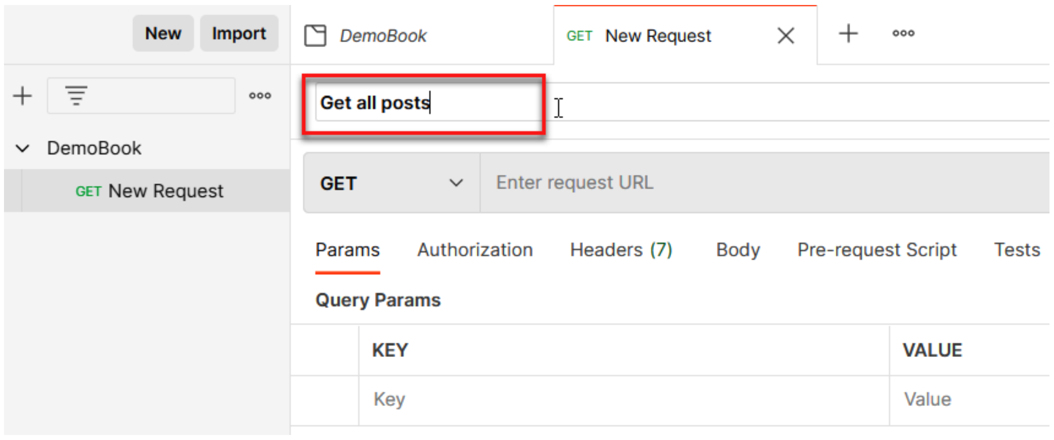


Figure 11.6 – Get all posts in Postman

2. **The configuration of the request**: After creating the request, we will configure it by entering the URL of the API to be tested, which is `https://jsonplaceholder.typicode.com/posts`, in the **GET** method. After entering the URL, we save the request configuration by clicking on the **Save** button.

The following screenshot shows the parameters of this request with its URL and method:

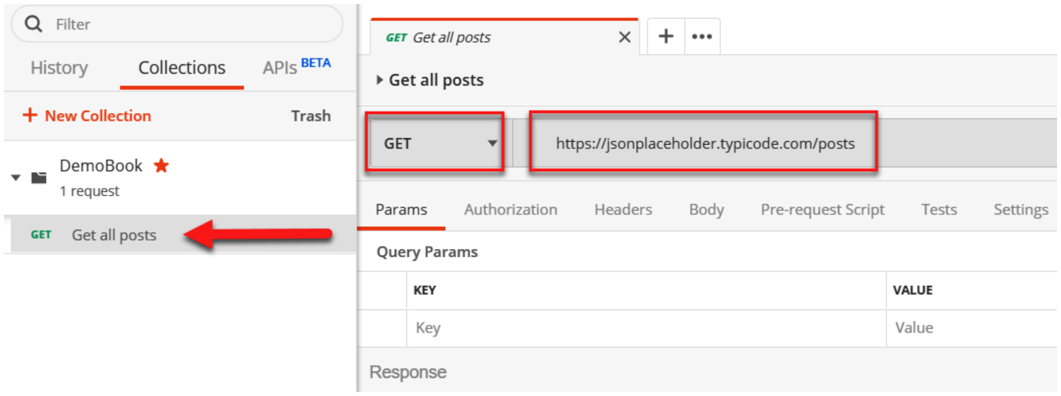


Figure 11.7 – Postman edit request

Finally, to complete the tests, and to add more content to our lab, we will add a second request to our collection, which we will call `Get a single post`. It will test another method of the API, and it will also ensure that we configure it with the `https://jsonplaceholder.typicode.com/posts/<ID of post>` URL.

The following screenshot shows the requests of our collection:

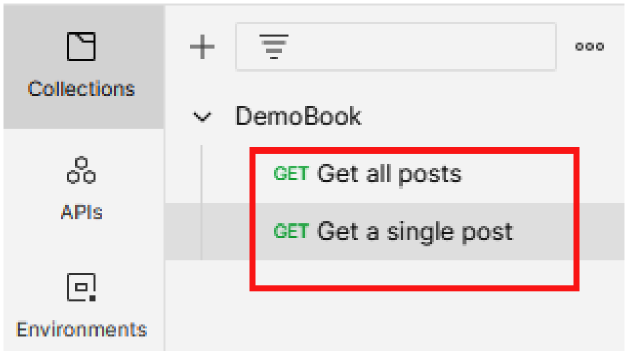


Figure 11.8 – Postman request list

Note

Note that the Postman documentation for collection creation can be found at `https://learning.getpostman.com/docs/postman/collections/creating_collections/`.

In this section, we have learned how to create a collection in Postman, as well as how to create requests and their configurations.

In the next section, we will learn how to dynamize our requests with the use of environments and variables.

Using environments and variables to dynamize requests

When we want to test an API, we need to test it on several environments for better results. For example, we will test it on our local machine and development environment, and then also on the QA environment. To optimize test implementation times and to avoid having a duplicate request in Postman, we will inject variables into this same request to make it testable in all environments.