**13 Dockerizing microservice APIs**

This chapter covers

* How to Dockerize an application
* How to run Docker containers
* How to run an application with Docker Compose
* Publishing a Docker image to AWS Elastic Container Registry

Docker is a virtualization technology that allows us to run our applications anywhere by simply having a Docker execution runtime. Docker takes away the pain and effort required to tune and configure an environment to run code. It also makes deployments more predictable since it produces replicable artifacts (container images) that we can run locally as well as in the cloud.

In this chapter, you’ll learn to Dockerize a Python application. Dockerizing is the process of packaging an application as a Docker image. You can think of a Docker image as a build or artifact that is ready to be deployed and executed. To execute an image, Docker creates running instances of the image, known as *containers*. To deploy Docker images, we typically use a container orchestrator, such as Kubernetes, which takes care of managing the life cycle of a container. In the next chapter, you’ll learn to deploy Docker builds with Kubernetes. We’ll illustrate how to Dockerize an application using the orders service of the CoffeeMesh platform. You’ll also learn to publish your Docker builds to a container registry by uploading images to AWS’s Elastic Container Registry (ECR).

All the code examples are available under folder ch13 in the GitHub repository for this book. We’ll begin by setting up the environment to work on this chapter in section 13.1.

**13.1 Setting up the environment for this chapter**

In this section, we set up the environment so that you can follow along with the examples in the rest of the chapter. We continue the implementation of the orders service where we left it in chapter 11, where we added the authentication and authorization layers. First, copy over the code from chapter 11 into a new folder called ch13:

$ cp -r ch11 ch13

cd into ch13, and install the dependencies and activate the virtual environment by running the following commands:

$ cd ch13 && pipenv install --dev && pipenv shell

When we deploy the application, we use a PostgreSQL engine, which is one of the most popular SQL engines for running applications in production. To communicate with the database, we use psycopg2, which is one of Python’s most popular PostgreSQL drivers:

$ pipenv install psycopg2

**INSTALLING PSYCOPG2** If you run into issues installing and compiling psycopg2, try installing the compiled package by running pipenv install psycopg2-binary, or pull ch13/Pipfile and ch13/Pipfile.lock from this book’s GitHub repository and run pipenv install --dev. Two other powerful PostgreSQL drivers are asyncpg (<https://github.com/MagicStack/asyncpg>) and pscycopg3 (<https://github.com/psycopg/psycopg>), both of which support asynchronous operations. I encourage you to check them out!

To build and run Docker containers, you’ll need a Docker runtime on your machine. Installation instructions are platform specific, so please see the official documentation to learn how to install Docker on your system (<https://docs.docker.com/get-docker/>).

Since we’re going to publish our Docker images to AWS’s ECR, we need to install the AWS CLI:

$ pipenv install --dev awscli

Next, go to <https://aws.amazon.com/>. Create an AWS account and obtain an access key to be able to access AWS services programmatically. The user profile you use to create the AWS account is the account’s root user. For security, it is recommended that you don’t use the root user to generate your access key. Instead, create an IAM user and generate an access key for that user. IAM is AWS’s Identity Access Management service, which allows you to create users, roles, and granular policies for granting access to other services in your account. Follow the AWS documentation to learn how to create an IAM user (<http://mng.bz/neP8>) and to learn how to generate your access keys and configure the AWS CLI (<http://mng.bz/vXxq>).

Now that our environment is ready, it’s time to Dockerize our applications!

**13.2 Dockerizing a microservice**

What does Dockerizing an application mean? *Dockerizing* is the process of packaging an application as a Docker image. You can think of a Docker image as a build or artifact that can be deployed and executed in a Docker runtime. All the system dependencies are already installed in the Docker image, and to run the image, we only need a Docker runtime. To execute the image, the Docker runtime creates a container, which is a running instance of the image. As you can see in figure 13.1, working with Docker is very convenient since it allows us to run our applications in isolated processes. There are different options for installing a Docker runtime depending on your platform, so please see the official documentation to determine which option works best for you (<https://docs.docker.com/get-docker/>).

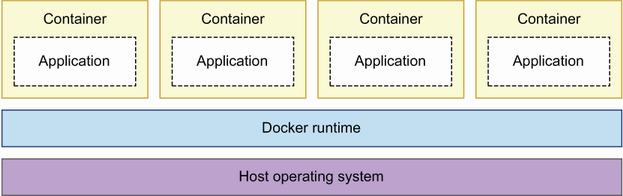


Figure 13.1 Docker containers run in isolated processes on top of the host operating system.

In this section, we create an optimized Docker image of the orders service. Along the way, you’ll learn how to write a Dockerfile, which is a document that contains all the instructions required to build a Docker image. You’ll also learn how to run Docker containers and to map ports from the container to the host operating system so that you can interact with the application running inside the container. Finally, you’ll also learn how to manage containers with the Docker CLI.

**DOCKER FUNDAMENTALS** If you want to know more about how Docker works and how it interacts with the host operating system, check out Prabath Siriwardena and Nuwan Dias’s excellent “Docker Fundamentals” from their book *Microservices Security in Action* (Manning, 2020, <http://mng.bz/49Ag>).

Before we build the image, we need to make two small changes to our application code to get it ready for deployment. So far, the orders service has been using a hardcoded database URL, but to operate the service in different environments, we need to make this setting configurable. The following code shows the changes needed to the orders/repository/unit\_of\_work.py file to pull the database URL from the environment, with the newly added code in bold characters. We use an assert statement to exit the application immediately if no database URL is provided.

Listing 13.1 Pulling the database URL from the environment

# file: orders/repository/unit\_of\_work.py

**import os**

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

**DB\_URL = os.getenv('DB\_URL')**  ①

**assert DB\_URL is not None, 'DB\_URL environment variable needed.'**  ②

class UnitOfWork:

def \_\_init\_\_(self):

self.session\_maker = sessionmaker(bind=create\_engine(**DB\_URL**)) ③

def \_\_enter\_\_(self):

self.session = self.session\_maker()

return self

...

① We pull the database URL from the DB\_URL environment variable.

② We exit the application if DB\_URL isn’t set.

③ We use the value from DB\_URL to connect to the database.

We also need to update our Alembic files to pull the database URL from the environment. The following code shows the changes required to migrations/env.py to accomplish that, with the newly added code in bold. We omitted nonrelevant parts of the code with ellipses to make it easier to observe the changes.

Listing 13.2 Pulling the database URL from the environment for alembic

# file: migrations/env.py

**import os**

from logging.config import fileConfig

**from sqlalchemy import create\_engine**

from sqlalchemy import pool

from alembic import context

...

def run\_migrations\_online():

"""...

"""

**url = os.getenv('DB\_URL')**  ①

**assert url is not None, 'DB\_URL environment variable needed.'** ②

connectable = create\_engine(url)

context.configure(

url=url,

target\_metadata=target\_metadata,

literal\_binds=True,

dialect\_opts={"paramstyle": "named"},

)

...

① We pull the database URL from the DB\_URL environment variable.

② We exit the application if DB\_URL isn’t set.

Now that our code is ready, it’s time to Dockerize it! To build a Docker image, we need to write a Dockerfile. Create a file named Dockerfile. Listing 13.3 shows this file’s contents. We use the slim version of the official Python 3.9 Docker image as our base image. Slim images contain just the dependencies that we need to run our applications, which results in lighter images. To use a base image, we use Docker’s FROM directive. Then we create the folder for the application code called /orders/orders. To run bash commands, such as mkdir in this case, we use Docker’s RUN directive. We also set /orders/orders as the working directory using Docker’s WORKDIR directive. The working directory is the directory from which the application runs.

Next, we install pipenv, copy our Pipenv files, and install the dependencies. We use Docker’s COPY directive to copy files from our filesystem into the Docker image. Since we’re running in Docker, we don’t need a virtual environment, so we install the dependencies using pipenv’s --system flag. We also use pipenv’s --deploy flag, which checks that our Pipenv files are up to date. Finally, we copy over our source code and specify the command that needs to be executed to get the orders service up and running. The command that Docker must use to execute our application is specified using Docker’s CMD directive. We also use Docker’s EXPOSE directive to make sure the running container listens on port 8000, the port on which our API runs. If we don’t expose the port, we can’t interact with the API.

The order of our statements in the Dockerfile matters because Docker caches each step of the build. Docker will only execute a step again if the previous step changed, for example, if we installed a new dependency, or if one of our files changed. Since our application code is likely to change more often than our dependencies, we copy the code at the end of the build. That way, Docker will only install the dependencies once and cache the step until they change.

Listing 13.3 Dockerfile for the orders service

# file: Dockerfile

FROM python:3.9-slim ①

RUN mkdir -p /orders/orders ②

WORKDIR /orders ③

RUN pip install -U pip && pip install pipenv

COPY Pipfile Pipfile.lock /orders/ ④

RUN pipenv install --system --deploy ⑤

COPY orders/orders\_service /orders/orders/orders\_service/ ⑥

COPY orders/repository /orders/orders/repository/

COPY orders/web /orders/orders/web/

COPY oas.yaml /orders/

COPY public\_key.pem /orders/public\_key.pem

COPY private.pem /orders/private.pem

EXPOSE 8000 ⑦

CMD ["uvicorn", "orders.web.app:app", "--host", "0.0.0.0"] ⑧

① The base image

② Base folder structure for our application

③ Working directory from which we’ll run the code

④ We copy our pipenv files.

⑤ We install the dependencies.

⑥ We copy the rest of the application files.

⑦ We expose the application’s port to the host machine.

⑧ The API server’s startup command

To build the Docker image from listing 13.3, you need to run the following command from the ch13 directory:

$ docker build -t orders:1.0 .

The -t flag stands for *tag*. A Docker tag has two parts: the image name on the left of the colon and the tag name on the right of the colon. The tag name is typically the version of the build. In this case, we’re naming the image orders and tagging it with 1.0. Make sure you don’t miss the period at the end of the build statement: it represents the path to the source code for the build (the *context* in Docker parlance). A period means the current directory.

Once the image has built, you can execute it with the following command:

$ docker run --env DB\_URL=sqlite:///orders.db \

-v $(pwd)/orders.db:/orders/orders.db -p 8000:8000 -it orders:1.0

As you can see in figure 13.2, the --env flag allows us to set environment variables in the container, and we use it to set the URL of the database. To make the application accessible to the host machine, we use the -p flag, which allows us to bind the port on which the application is running inside the container to a port in the host machine. We also use the -v flag to mount a volume on the SQLite database file. Docker volumes allow containers to access files from the host machine’s file system.

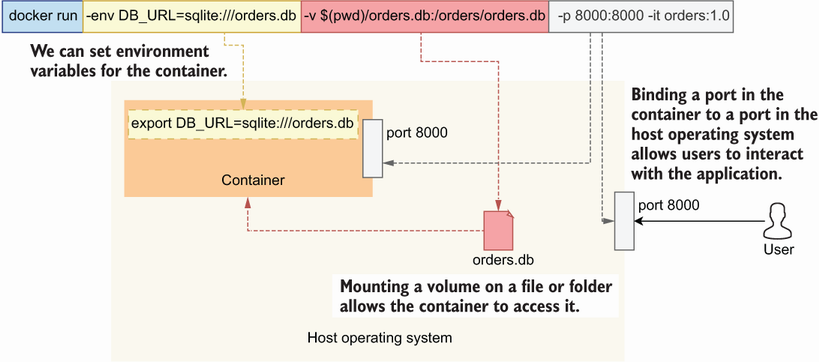


Figure 13.2 When we run a container, we can include various configurations to set environment variables within the container or to allow it to access files in the host operating system.

You can now access the application on the following URL: http://127.0.0.1:8000/docs/orders. The previous command executes the container attached to your current terminal session, which allows you to see the logs unfold as you interact with the application. In this case, you can stop the container just like any other process by pressing the Ctrl-C key combination.

You can also run containers in detached mode, which means the process isn’t linked to your terminal session, so when you close your terminal, the process will continue running. This is convenient if you just want to run a container to interact with it, and you don’t need to watch the logs. We typically run containerized databases in detached mode. To run the container in detached mode, you use the -d flag:

$ docker run **-d** –-env DB\_URL=sqlite:///orders.db \

-v $(pwd)/orders.db:/orders/orders.db -p 8000:8000 orders:1.0

In this case, you’ll need to stop the container with the docker stop command. First, you need to figure out the ID of the running container with the following command:

$ docker ps

This command will list all currently running containers in your machine. The output looks like this (output truncated with ellipses):

CONTAINER ID IMAGE COMMAND CREATED STATUS...

83e6189a02ee orders:1.0 "uvicorn..." 7 seconds ago Up 6 seconds

Pick up the container ID (in this case 83e6189a02ee) and use it to stop the process with the following command:

$ docker stop 83e6189a02ee

That’s all it takes to build and run Docker containers! There’s a lot more to Docker than we’ve seen in this section, and if you’re interested in learning more about this technology, I recommend you look at *Docker in Practice* by Ian Miell and Aidan Hobson Sayers (Manning, 2019) and *Docker in Action* by Jeff Nickoloff and Stephen Kuenzli (Manning, 2019).

**13.3 Running applications with Docker Compose**

In the previous section, we ran the orders service’s container by mounting it on our local SQLite database. This is fine for a quick test, but it doesn’t really tell us whether our application will work as expected with a PostgreSQL database. A common strategy to connect our containerized applications to a database is using Docker Compose, which allows us to run multiple containers within a shared network, so they can talk to each other. In this section, you’ll learn how to run the orders service with a PostgreSQL database using docker-compose.

To use Docker Compose, first we need to install it. It is a Python package, so we install it with pip:

$ pip install docker-compose

Next, let’s write our Docker Compose file—a declaration of the resources we need to run our application. Listing 13.4 shows the docker-compose file for the orders service. We use Docker Compose’s latest specification format, version 3.9, and we declare two services: database and api. database runs PostgreSQL’s official Docker image, while api runs the orders service. We use the build keyword to point to the Docker build context, and we give it a period value (.). By using a period, we instruct Docker Compose to look for a Dockerfile and build the image relative to the current directory. Through the environment keyword, we configure the environment variables required to run our applications. We expose database’s 5432 port so that we can connect to the database from our host machine, as well as api’s 8000 port so that we can access the API. Finally, we use a volume called database-data, which docker-compose will use to persist our data. This means that if you restart docker-compose, you won’t lose your data.

Listing 13.4 docker-compose file for the orders service

# file: docker-compose.yaml

version: "3.9" ①

services: ②

database: ③

image: postgres:14.2 ④

ports: ⑤

- 5432:5432

environment: ⑥

POSTGRES\_PASSWORD: postgres

POSTGRES\_USER: postgres

POSTGRES\_DB: postgres

volumes: ⑦

- database-data:/var/lib/postgresql/data

api: ⑧

build: . ⑨

ports: ⑩

- 8000:8000

depends\_on: ⑪

- database

environment: ⑫

DB\_URL: postgresql://postgres:postgres@database:5432/postgres

volumes: ⑬

database-data:

① The version of docker-compose’s format for this file.

② We declare our services.

③ The database service

④ The database service’s Docker image

⑤ We expose the database ports to the host machine.

⑥ Database environment configuration

⑦ We mount our database’s data folder on a local volume.

⑧ The API service

⑨ The API’s build context

⑩ We expose the API’s port to the host machine.

⑪ The API depends on the database.

⑫ The API’s environment configuration

⑬ The database’s volume

Execute the following command to run our Docker Compose file:

$ docker-compose up --build

The --build flag instructs Docker Compose to rebuild your images if your files changed. Once the web API is up and running, you can access it on http://localhost: 8000/docs/orders. If you try any of the endpoints, your tables don’t exist. That’s because we haven’t run the migrations against our fresh PostgreSQL database! To run the migrations, open a new terminal window, cd into the ch13 folder, activate your pipenv environment, and run the following command:

$ PYTHONPATH=`pwd` \

DB\_URL=postgresql://postgres:postgres@localhost:5432/postgres alembic \

upgrade heads

Once the migrations have been applied, you can hit the API endpoints again, and everything should work. To stop docker-compose, run the following command from another terminal window and inside the ch13 folder:

$ docker-compose down

This is all it takes to run Docker Compose! You’ve just learned to use one of the most powerful automation tools. Docker Compose is often used to run integration tests and to have an easy way to run the backend for developers who work on client applications, such as SPAs.

With our Docker stack ready and our images tested, it’s time to learn how to push images to a container registry. Move on to the next section to learn how!

**13.4 Publishing Docker builds to a container registry**

To deploy our Docker builds, we need to publish them first to a Docker container registry. A container registry is a repository of Docker images. In the next chapter, we will deploy our applications to AWS’s Elastic Kubernetes Service, so we publish our builds to AWS’s ECR. Keeping our Docker images within AWS will make it easier to deploy them to EKS.

First, let’s create an ECR repository for our images with the following command:

$ aws ecr create-repository --repository-name coffeemesh-orders

{

"repository": {

**"repositoryArn":**

➥ **"arn:aws:ecr:<aws\_region>:<aws\_account\_id>:repository/coffeemesh-orders",**

"registryId": "876701361933",

"repositoryName": "coffeemesh-orders",

"repositoryUri":

➥ "<aws\_account\_id>.dkr.ecr.<aws\_region>.amazonaws.com/coffeemesh-orders",

"createdAt": "2021-11-16T10:08:42+00:00",

"imageTagMutability": "MUTABLE",

"imageScanningConfiguration": {

"scanOnPush": false

},

"encryptionConfiguration": {

"encryptionType": "AES256"

}

}

}

In this command, we create a ECR repository named coffeemesh-orders. The output from the command is a payload describing the repository we just created. When you run the command, the placeholder for <aws\_account\_id> in the output payload will contain your AWS account ID, and <aws\_region> will contain your default AWS region. To publish our Docker build to ECR, we need to tag our build with the name of the ECR repository. Get hold of the repository.repositoryArn property of the previous command’s output (in bold), and use it to tag the Docker build we created in section 13.2 with the following command:

$ docker tag orders:1.0 \

<aws\_account\_id>.dkr.ecr.<aws\_region>.amazonaws.com/coffeemesh-orders:1.0

To publish our images to ECR, we need to obtain login credentials with the following command:

$ aws ecr get-login-password --region <aws\_region> | docker login \

--username AWS --password-stdin \

<aws\_account\_id>.dkr.ecr.<region>.amazonaws.com

Make sure you replace <aws\_region> in this command for the AWS region where you created the Docker repository, such as eu-west-1 for Europe (Ireland) or us-east-2 for US East (Ohio). Also replace <aws\_account\_id> with your AWS account ID. Check out the AWS documentation to learn how to find your AWS account ID (<http://mng.bz/Qnye>).

**AWS REGIONS** When you deploy services to AWS, you deploy them to specific regions. Each region has an identifier, such as eu-west-1 for Ireland and eu-east-2 for Ohio. For an up-to-date list of the regions available in AWS, see <http://mng.bz/XaPM>.

The aws ecr get-login-password command produces an instruction that Docker knows how to use to log in to ECR. We’re now ready to publish our build! Run the following command to push the image to ECR:

$ docker push \

<aws\_account\_id>.dkr.ecr.<aws\_region>.amazonaws.com/coffeemesh-orders:1.0

Voila! Our Docker build is now in ECR. In the next chapter, you’ll learn how to deploy this build to a Kubernetes cluster in AWS.

**Summary**

* Docker is a virtualization technology that allows us to run our applications anywhere by simply having a Docker execution runtime. A Docker build is called an image, which is executed in processes called Docker containers.
* Docker Compose is a container orchestration framework that allows you to run multiple containers simultaneously, such as databases and APIs. Using Docker Compose is an easy and effective way to run your whole backend without having to install and configure additional dependencies.
* To deploy Docker images, we publish them to a container registry, such as AWS’s ECR—a robust and secure container registry that makes it easy to deploy our containers to AWS services.