Additionally, using a Dockerfile and a docker-compose file eliminates the need to upload and share images of our applications. New versions of our applications can be built from the Dockerfile and deployed using the docker-compose file. Application images can also be stored and retrieved from **Docker Hub**. This is known as a push and pull operation.

To begin setting up, download and install Docker from `https://docs.docker.com/install`.

Dockerfile

A Dockerfile contains instructions on how our application image is to be built. The following is an example Dockerfile:

```
FROM PYTHON:3.8
# Set working directory to /usr/src/app
WORKDIR /usr/src/app
# Copy the contents of the current local directory into the
container's working directory
ADD . /usr/src/app
# Run a command
CMD ["python", "hello.py"]
```

Next, we'll build the application container image and tag it `getting_started` as follows:

```
$ docker build -t getting_started .
```

If the Dockerfile isn't present in the directory where the command is being run, the path to the Dockerfile should be properly appended as follows:

```
$ docker build -t api api/Dockerfile
```

The container image can be run using the following command:

```
$ docker run getting-started
```

Docker is an efficient tool for containerization. We have only looked at the basic operations and we'll learn more operations practically in *Chapter 9*, *Deploying FastAPI Applications*.

Building a simple FastAPI application

Finally, we can now get to our first FastAPI project. Our aim in this section is to introduce FastAPI by building a simple application. We shall cover in-depth operations in subsequent chapters.

We'll begin by installing the dependencies required for our application in the `todos` folder we created earlier. The dependencies are the following:

- `fastapi`: The framework on which we'll build our application.
- `uvicorn`: An Asynchronous Server Gateway Interface module to run our application.

First, activate your development environment by running the following command in your project directory:

```
$ source venv/bin/activate
```

Then, install the dependencies as follows:

```
(venv)$ pip install fastapi uvicorn
```

For now, we'll create a new `api.py` file and create a new instance of FastAPI as follows:

```
from fastapi import FastAPI

app = FastAPI()
```

By instantiating FastAPI in the app variable, we can proceed to create routes. Let's create a welcome route.

A route is created by first defining a decorator to indicate the type of operation, followed by a function containing the operation to be carried out when this route is invoked. In the following example, we'll create a `"/"` route that only accepts `GET` requests and returns a welcome message when visited:

```
@app.get("/")
async def welcome() -> dict:
    return { "message": "Hello World"}
```

The next step is to start our application using `uvicorn`. In your terminal, run the following command:

```
(venv)$ uvicorn api:app --port 8000 --reload
```

In the preceding command, `uvicorn` takes the following arguments:

- `file:instance`: The file containing the instance of FastAPI and the name variable holding the FastAPI instance.
- `--port PORT`: The port the application will be served on.
- `--reload`: An optional argument included to restart the application on every file change.

The command returns the following output:

```
(venv) ➜  todos uvicorn api:app --port 8080 --reload
INFO:     Will watch for changes in these directories: ['/
Users/youngestdev/Documents/todos']
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C
to quit)
INFO:     Started reloader process [3982] using statreload
INFO:     Started server process [3984]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

The next step is to test the application by sending a `GET` request to the API. In a new terminal, send a `GET` request using `curl` as follows:

```
$ curl http://0.0.0.0:8080/
```

The response from the application logged in your console will be the following:

```
{"message":"Hello World"}
```

Summary

In this chapter, we have learned how to install the tools required to set up our development environment. We have also built a simple API as an introduction to FastAPI and learned how to create a route in the process.

In the next chapter, you will be introduced to routing in FastAPI. First, you will be introduced to the process of building models to validate request payloads and responses using Pydantic. You will then learn about Path and Query parameters as well as request body, and finally, you will learn how to build a CRUD todo application.

2
Routing in FastAPI

Routing is an essential part of building a web application. Routing in FastAPI is flexible and hassle-free. Routing is the process of handling **HTTP requests** sent from a client to the server. HTTP requests are sent to defined routes, which have defined handlers for processing the requests and responding. These handlers are called route handlers.

By the end of this chapter, you will know how to create routes using the **APIRouter** instance and connect to the main **FastAPI** application. You will also learn what models are and how to use them to validate request bodies. You will also learn what path and query parameters are and how to use them in your FastAPI application. The knowledge of routing in FastAPI is essential in building small- and large-scale applications.

In this chapter, we'll be covering the following topics:

- Routing in FastAPI
- The `APIRouter` class
- Validation using Pydantic models
- Path and query parameters
- Request body
- Building a simple CRUD app

Technical requirements

The code used in this chapter can be found at the `https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch02/todos`.

Understanding routing in FastAPI

A route is defined to accept requests from an HTTP request method and optionally take parameters. When a request is sent to a route, the application checks whether the route is defined before processing the request in the route handler. On the other hand, a route handler is a function that processes the request sent to the server. An example of a route handler is a function that retrieves records from a database when a request is sent to a router via a route.

> **What are HTTP request methods?**
>
> HTTP methods are identifiers for indicating the type of action to be carried out. The standard methods include `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`. You can learn more about HTTP methods at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.

Routing example

In the *Project scaffolding* section in the previous chapter, we built a single route application. The routing was handled by the `FastAPI()` instance initiated in the `app` variable:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def welcome() -> dict:
    return { "message": "Hello World"}
```

The **uvicorn** tool was pointed to the FastAPI's instance to serve the application:

```
(venv)$ uvicorn api:app --port 8080 --reload
```