Traditionally, the `FastAPI()` instance can be used for routing operations, as seen previously. However, this method is commonly used in applications that require a single path during routing. In a situation where a separate route performing a unique function is created using the `FastAPI()` instance, the application will be unable to run both routes, as uvicorn can only run one entry point.

How then do you handle extensive applications that require a series of routes performing different functions? We'll look at how the **APIRouter** class helps with multiple routing in the next section.

Routing with the APIRouter class

The `APIRouter` class belongs to the FastAPI package and creates path operations for multiple routes. The `APIRouter` class encourages modularity and organization of application routing and logic.

The `APIRouter` class is imported from the `fastapi` package, and an instance is created. The route methods are created and distributed from the instance created, such as the following:

```
from fastapi import APIRouter

router = APIRouter()

@router.get("/hello")
async def say_hello() -> dict:
    return {"message": "Hello!"}
```

Let's create a new path operation with the `APIRouter` class to create and retrieve todos. In the `todos` folder from the previous chapter, create a new file, `todo.py`:

```
(venv)$ touch todo.py
```

We'll start by importing the `APIRouter` class from the fastapi package and creating an instance:

```
from fastapi import APIRouter

todo_router = APIRouter().
```

Next, we'll create a temporary in-app database, alongside two routes for the addition and retrieval of todos:

```
todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: dict) -> dict:
    todo_list.append(todo)
    return {"message": "Todo added successfully"}


@todo_router.get("/todo")
async def retrieve_todos() -> dict:
    return {"todos": todo_list}
```

In the preceding code block, we have created two routes for our todo operations. The first route adds a todo to the todo list via the `POST` method, and the second route retrieves all the todo items from the todo list via the `GET` method.

We have completed the path operations for the todo route. The next step is to serve the application to production so that we can test the path operations defined.

The `APIRouter` class works in the same way as the `FastAPI` class does. However, uvicorn cannot use the `APIRouter` instance to serve the application, unlike the FastAPIs. Routes defined using the APIRouter class are added to the fastapi instance to enable their visibility.

To enable the visibility of the todo routes, we'll include the `todo_router` path operations handler to the primary `FastAPI` instance using the `include_router()` method.

> **include_router()**
>
> The `include_router(router, ...)` method is responsible for adding routes defined with the `APIRouter` class to the main application's instance to enable the routes to become visible.

In `api.py`, import `todo_router` from `todo.py`:

```
from todo import todo_router
```

Include the `todo_router` in the FastAPI application, using the `include_router` method from the **FastAPI** instance:

```
from fastapi import FastAPI
from todo import todo_router

app = FastAPI()


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

app.include_router(todo_router)
```

With everything in place, start the application from your terminal:

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

The preceding command starts our application and gives us a real-time log of our application processes:

```
(venv) ➜  todos git:(main) ✗ uvicorn api:app --port 8000
--reload
INFO: Will watch for changes in these directories: ['/Users/
youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/
ch02/todos']
INFO:     uvicorn running on http://127.0.0.1:8000 (Press
CTRL+C to quit)
INFO:     Started reloader process [4732] using statreload
INFO:     Started server process [4734]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

The next step is to test the application by sending a `GET` request using `curl`:

```
(venv)$ curl http://0.0.0.0:8080/
```

The response from the application logged in your console:

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

Next, we check whether the todo routes are functional:

```
(venv)$ curl -X 'GET' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json'
```

The response from the application logged in your console should be as follows:

```
{
  "todos": []
}
```

The todo route worked! Let's test the `POST` operation by sending a request to add an item to our todo list:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 1,
  "item": "First Todo is to finish this book!"
}'
```

We have the following response:

```
{
  "message": "Todo added successfully."
}
```

We've learned how the `APIRouter` class works and how to include it in the primary application instance to enable the usage of the path operations defined. The todo routes built in this section lacked models, otherwise known as schemas. In the next section, let's take a look at **Pydantic** models and their use cases.

Validating request bodies using Pydantic models

In FastAPI, request bodies can be validated to ensure only defined data is sent. This is crucial, as it serves to sanitize request data and reduce malicious attacks' risks. This process is known as validation.

A model in FastAPI is a structured class that dictates how data should be received or parsed. Models are created by subclassing Pydantic's `BaseModel` class.

> **What is Pydantic?**
>
> Pydantic is a Python library that handles data validation using Python-type annotations.

Models, when defined, are used as type hints for request body objects and request-response objects. In this chapter, we will only look at using Pydantic models for request bodies.

An example model is as follows:

```
from pydantic import BaseModel

class PacktBook(BaseModel):
    id: int
    Name: str
    Publishers: str
    Isbn: str
```

In the preceding code block above, we defined a `PacktBook` model as a subclass of Pydantic's `BaseModel` class. A variable type hinted to the `PacktBook` class can only take four fields, as defined previously. In the next couple of examples, we see how Pydantic helps in validating inputs.

In our todo application earlier, we defined a route to add an item to our `todo` list. In the route definition, we set the request body to a dictionary:

```
async def add_todo(todo: dict) -> dict:
   …
```