Technical requirements

The code used in this chapter can be found at `https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch03/todos`.

Understanding responses in FastAPI

Responses are an integral part of an API's life cycle. Responses are the feedback received from interacting with an API route via any of the standard HTTP methods. An API response is usually in JSON or XML format, but it can also be in the form of a document. A response consists of a header and a body.

What is a response header?

A response header consists of the request's status and additional information to guide the delivery of the response body. An example of the information contained in the response header is `Content-Type`, which tells the client the content type returned.

What is a response body?

The response body, on the other hand, is the data requested from the server by the client. The response body is determined from the `Content-Type` header variable and the most commonly used one is `application/json`. In the previous chapter, the list of to-dos returned is the response body.

Now that you've learned what responses are and what they consist of, let's take a look at HTTP status codes included in responses in the next section.

Status codes

Status codes are unique short codes issued by a server in response to a client's request. Response status codes are grouped into five categories, each denoting a different response:

- `1XX`: Request has been received.
- `2XX`: The request was successful.
- `3XX`: Request redirected.
- `4XX`: There's an error from the client.
- `5XX`: There's an error from the server.

A complete list of HTTP status codes can be found at `https://httpstatuses.com/`.

The first digit of a status code defines its category. Common status codes include `200` for a successful request, `404` for request not found, and `500` indicating an internal server error.

The standard practice followed in building web applications, irrespective of the framework, is to return appropriate status codes for individual events. A `400` status code shouldn't be returned for a server error. Likewise, a `200` status code shouldn't be returned for a failed request operation.

Now that you have learned what status codes are, let's learn how to build response models in the next section.

Building response models

We established the purpose of response models at the beginning of this chapter. You also learned how to build models in the previous chapter using Pydantic. Response models are also built on Pydantic but serve a different purpose.

In the definition of route paths, we have the following, for example:

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

The route returns a list of to-dos present in the database. Here's some example output:

```
{
  "todos": [
    {
      "id": 1,
      "item": "Example schema 1!"
    },
    {
      "id": 2,
      "item": "Example schema 2!"
    },
    {
      "id": 3,
```

```
        "item": "Example schema 5!"
      }
    ]
}
```

The route returns all the content stored in the todos array. To specify the information to be returned, we would have to either separate data to be displayed or introduce additional logic. Fortunately, we can create a model containing the fields we want to be returned and add it to our route definition using the `response_model` argument.

Let's update the route that retrieves all the to-dos to return an array of just the to-do items and not the IDs. Let's start by defining a new model class to return a list of to-do items in `model.py`:

```
from typing import List

class TodoItem(BaseModel):
    item: str

    class Config:
        schema_extra = {
            "example": {
                "item": "Read the next chapter of the book"
            }
        }

class TodoItems(BaseModel):
    todos: List[TodoItem]

    class Config:
        schema_extra = {
            "example": {
                "todos": [
                    {
                        "item": "Example schema 1!"
                    },
                    {
                        "item": "Example schema 2!"
```

```
                    }
                ]
            }
        }
```

In the preceding code block, we have defined a new model, `TodoItems`, which returns a list of variables contained in the `TodoItem` model. Let's update our route in `todo.py` by adding a response model to it:

```
from model import Todo, TodoItem, TodoItems
…
@todo_router.get("/todo", response_model=TodoItems)
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }
```

Activate your virtual environment and start your application:

```
$ source venv/bin/activate
(venv)$ uvicorn api:app --host=0.0.0.0 --port 8000 --reload
```

Next, add a new to-do:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 1,
  "item": "This todo will be retrieved without exposing my
   ID!"
}'
```

Retrieve the to-dos:

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

The response received is as follows:

```
{
  "todos": [
    {
      "item": " This todo will be retrieved without
       exposing my ID!"
!"
    }
  ]
}
```

Now that we have learned what response models are and how to use them, we will continue to use them where they fit in subsequent chapters. Let's take a look at error responses and how to handle errors in the next section.

Error handling

Earlier on in this chapter, we learned what status codes are and how they are useful in informing the client about the request status. Requests can return erroneous responses, and these responses can be ugly or have insufficient information about the cause of failure.

Errors from requests can result from attempting to access non-existent resources, protected pages without sufficient permissions, and even server errors. Errors in FastAPI are handled by raising an exception using FastAPI's `HTTPException` class.

> **What Is an HTTP Exception?**
>
> An HTTP exception is an event that is used to indicate a fault or issue in the request flow.