In the example `POST` request, the data sent was in the following format:

```
{
   "id": id,
   "item": item
}
```

However, an empty dictionary could've also been sent without returning any error. A user can send a request with a body different from the one shown previously. Creating a model with the required request body structure and assigning it as a type to the request body ensures that only the data fields present in the model are passed.

For example, to ensure only the request body contains fields in the preceding example, create a new `model.py` file and add the following code below to it:

```
from Pydantic import BaseModel

class Todo(BaseMode):
    id: int
    item: str
```

In the preceding code block, we have created a Pydantic model that accepts only two fields:

- `id` of type integer
- `item` of type string

Let's go ahead and use the model in our `POST` route. In `api.py`, import the model:

```
from model import Todo
```

Next, replace the request body variable type from `dict` to `Todo`:

```
todo_list = []

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

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

Let's verify the new request body validator by sending an empty dictionary as the request body:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
}'
```

We get a response, indicating the absence of the `id` and `item` field in the request body:

```
{
  "detail": [
    {
      "loc": [
        "body",
        "id"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    },
    {
      "loc": [
        "body",
        "item"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

Sending a request with correct data returns a successful response:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 2,
  "item": "Validation models help with input types"
}'
```

Here is the response:

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

Nested models

Pydantic models can also be nested, such as the following:

```
class Item(BaseModel)
    item: str
    status: str

class Todo(BaseModel)
    id: int
    item: Item
```

As a result, a todo of type `Todo` will be represented as the following:

```
{
  "id": 1,
  "item": {
        "item": "Nested models",
        "Status": "completed"
    }
}
```

We have learned what models are, how to create one, and their use cases. We will be using it subsequently in the remaining parts of this book. In the next section, let's look at **path** and **query** parameters.

Path and query parameters

In the previous section, we learned what models are and how they are used to validate request bodies. In this section, you'll learn what path and query parameters are, the role they play in routing, and how to use them.

Path parameters

Path parameters are parameters included in an API route to identify resources. These parameters serve as an identifier and, sometimes, a bridge to enable further operations in a web application.

We currently have routes for adding a todo and retrieving all the todos in our todo application. Let's create a new route for retrieving a single todo by appending the todo's ID as a path parameter.

In `todo.py`, add the new route:

```
from fastapi import APIRouter, Path
from model import Todo

todo_router = APIRouter()

todo_list = []


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

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



@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The
ID of the todo to retrieve.")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return {
                "todo": todo
            }
    return {
        "message": "Todo with supplied ID doesn't exist."
    }
```

In the preceding code block, `{todo_id}` is the path parameter. This parameter enables the application to return a matching todo with the ID passed.

Let's test the route:

```
(venv)$ curl -X 'GET' \
  'http://127.0.0.1:8000/todo/1' \
  -H 'accept: application/json'
```

In the preceding `GET` request, `1` is the path parameter. Here, we are telling our todo application to return the `todo` item with `1` ID.

Executing the preceding request results in the following response:

```
{
  "todo": {
    "id": 1,
    "item": "First Todo is to finish this book!"
  }
}
```