

The `HTTPException` class takes three arguments:

- `status_code`: The status code to be returned for this disruption
- `detail`: Accompanying message to be sent to the client
- `headers`: An optional parameter for responses requiring headers

In our to-do route path definitions, we return a message when a to-do can't be found. We will be updating it to raise `HTTPException`. `HTTPException` allows us to return an adequate error response code.

In our current application, retrieving a to-do that doesn't exist returns a 200 response status code instead of a 404 response status code on `http://127.0.0.1:8000/docs`:

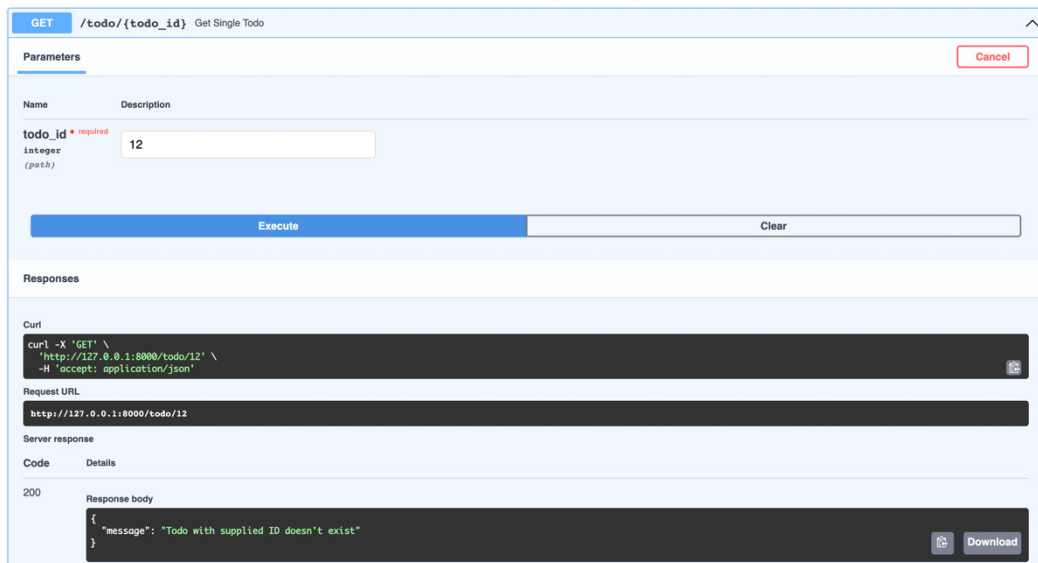


Figure 3.1 – Request returns a 200 response instead of a 404 response

By updating the routes to use the `HTTPException` class, we can return relevant details in our response. In `todo.py`, update the routes for retrieving, updating, and deleting a to-do:

```
from fastapi import APIRouter, Path, HTTPException, status
...
@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
            }
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Todo with supplied ID doesn't exist",
        )

@todo_router.put("/todo/{todo_id}")
async def update_todo(todo_data: TodoItem, todo_id: int =
    Path(..., title="The ID of the todo to be updated.)) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            todo.item = todo_data.item
            return {
                "message": "Todo updated successfully."
            }

        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Todo with supplied ID doesn't exist",
        )

@todo_router.delete("/todo/{todo_id}")
async def delete_single_todo(todo_id: int) -> dict:
    for index in range(len(todo_list)):
        todo = todo_list[index]
        if todo.id == todo_id:
            todo_list.pop(index)
            return {
                "message": "Todo deleted successfully."
            }

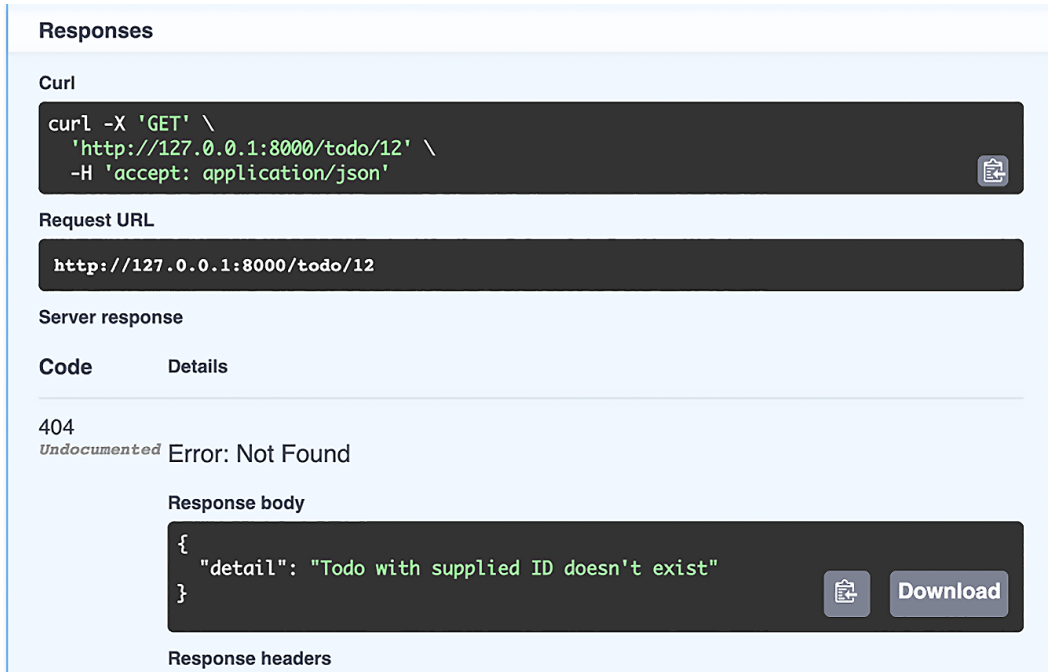
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
```

```

        detail="Todo with supplied ID doesn't exist",
    )

```

Now, let's retry retrieving the non-existent to-do to verify that the right response code is returned:



**Responses**

**Curl**

```
curl -X 'GET' \
'http://127.0.0.1:8000/todo/12' \
-H 'accept: application/json'
```

**Request URL**

```
http://127.0.0.1:8000/todo/12
```

**Server response**

| Code | Details                       |
|------|-------------------------------|
| 404  | Undocumented Error: Not Found |

**Response body**

```
{
  "detail": "Todo with supplied ID doesn't exist"
}
```

**Response headers**

Figure 3.2 – The correct 404 response code displayed

Lastly, we can declare the HTTP status code to override the default status code for successful operations by adding the `status_code` argument to the decorator function:

```

@todo_router.post("/todo", status_code=201)
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }

```

We have learned how to return the right response codes to clients, as well as overriding the default status code, in this section. It is also important to note that the default status code for success is 200.

## Summary

In this chapter, we learned what responses and response models are and what is meant by error handling. We also learned about HTTP status codes and why it is important to use them.

We also created response models from the knowledge we gained about creating models from the previous chapter and created a response model to return only the items in the to-do list without their IDs. Lastly, we learned about errors and error handling. We updated our existing routes to return the right response code instead of the default 200 status code.

In the next chapter, you will be introduced to templating FastAPI applications with Jinja. You will first be introduced to the basics needed to get you up and running with Jinja templating, after which you will create a user interface using your templating knowledge for our simple to-do application.

# 4

# Templating in FastAPI

Now that we have learned how to handle responses from requests including errors in the previous chapter, we can proceed to render the responses from the request on a web page. In this chapter, we will learn how to render responses from our API to a web page using templates powered by **Jinja**, which is a templating language written in Python designed to help the rendering process of API responses.

Templating is the process of displaying the data gotten from the API in various formats. Templates act as a frontend component in web applications.

By the end of this chapter, you will be equipped with the knowledge of what templating is and how to use templates to render information from your API. In this chapter, we'll be covering the following topics:

- Understanding Jinja
- Using Jinja2 templates in FastAPI

## Technical requirements

The code used in this chapter can be found at <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch04/todos>.