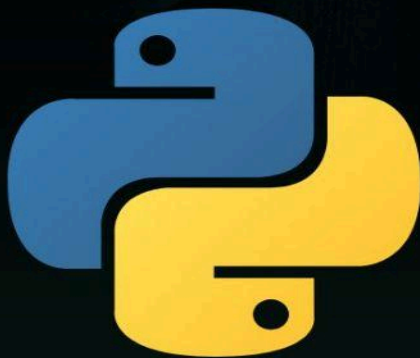




MASTERING

 FastAPI



BY ISMAIL CISSE

**Copyright © 2024 Ismail Cisse
All rights reserved.**

Unauthorized reproduction, distribution, or transmission of this eBook, or any portion thereof, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.

Violators will be prosecuted and may face legal action, including but not limited to, financial damages and other remedies as deemed appropriate by the courts.

Table of Contents

Introduction to FastAPI	5
What is Json	6
FastAPI Fundamentals	7
Path parameter	9
Posting request using Postman	10
CRUD OPERATIONS USING FASTAPI	13
Request Body and Post Method	15
Deleting an item	22
Status Codes	23

Introduction to FastAPI

Imagine FastAPI is like a magical chef 🧑🍳 just like gundulf the greasy in your kitchen. You tell this chef what you want to eat (your request), and he quickly whips it up for you, whether it's a sandwich, soup, or a three-course meal (your response). FastAPI is a modern web framework for building APIs with Python, allowing you to create APIs very fast using just a few lines of code. It's super fast, just like a magical chef who can cook at lightning speed, and it's very easy to use.

You can tell FastAPI what ingredients (data) you have and what dish (API) you want to create, and it handles everything else for you. Everything is automatically done for you, including data validation, auto-documentation, auto-completion, and suggestions.

What is an api , if you don't have a basics of an api I suggest you download the following short books that explain to you in an easy and funny way to understand an api and help you create your own api in less than 20 minutes click the link to download:

<https://github.com/Smailya/API/blob/main/pythonApi/LEARNANDCREATEYOURFIRSTAPIUSINGPYTHONANDFLASK.pdf>

What is Json

Imagine you're at your favorite burrito joint. You've got a tortilla (that's your data container), and you need to fill it with all your favorite ingredients like beans, rice, cheese, guac, and salsa (those are your data items). Once it's all wrapped up, you've got a neat, delicious burrito (a JSON object)!

JSON stands for JavaScript Object Notation is like a burrito for your data! It's a way to pack information in a way that's easy to transport and understand. Just like a burrito, you can take it anywhere, and everyone knows how to unwrap it and see what's inside.

JSON Ingredients:

- Keys and Values: Each ingredient (key) has its amount (value).
- Braces: {}: These are like the tortilla wraps.
- Colons and Commas: : and , : These are like the sauces and spices, making everything fit together nicely.

Example of JSON

```
json
{
  "name": "Veggie Burrito",
  "ingredients": ["beans", "rice", "cheese", "guac", "salsa"],
  "price": 7.99
}
```

FastAPI is like your favorite food delivery app, but for data burritos. It helps you send and receive these JSON burritos between different places (servers and clients) efficiently and quickly.

Let's say you're running a burrito shop, and you need to send a burrito order to the kitchen (API) and get a burrito back when it's ready.

1. Create Your Burrito Order: Make a JSON object (the order).

2. **Send It to the Kitchen:** Use FastAPI to send this JSON order to the server.
3. **Receive the Burrito:** Get a JSON response back from the server with your delicious burrito details.

Download visual studio code as your editor:

<https://code.visualstudio.com/download>

You can download the source code below:

<https://github.com/Smailya/API/tree/main/FASTAPI>

FastAPI Fundamentals

Creating Your First FastAPI Application

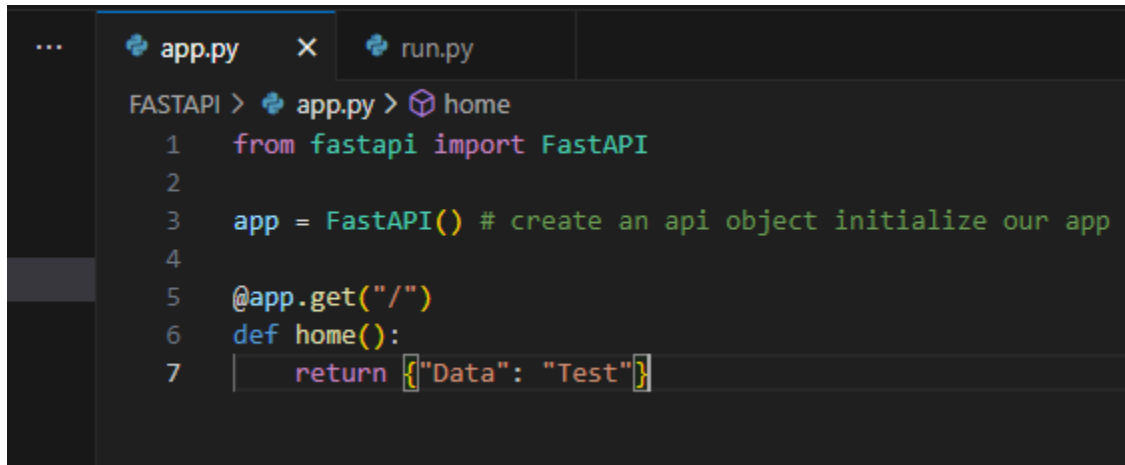
Create a Python file and name it app.py:

Installing FastAPI and Uvicorn

Open cmd from where the app.py location To install FastAPI, type:

pip install fastapi

Then, install Uvicorn, which is like a web server: pip install uvicorn

A screenshot of a code editor with a dark theme. The editor has two tabs at the top: 'app.py' (active) and 'run.py'. The code in 'app.py' is as follows:

```
FASTAPI > app.py > home
1  from fastapi import FastAPI
2
3  app = FastAPI() # create an api object initialize our app
4
5  @app.get("/")
6  def home():
7      return {"Data": "Test"}
```

Here is the explanation of the code below:

Import the FastAPI class from the fastapi module.

```
from fastapi import FastAPI
```

Create an instance of the FastAPI class. This instance will be our main application object.

```
app = FastAPI() # Initialize the FastAPI app
```

Define a route using the @app.get decorator. This route will handle GET requests to the root URL ("/).

```
@app.get("/") # Define a route for the root URL
```

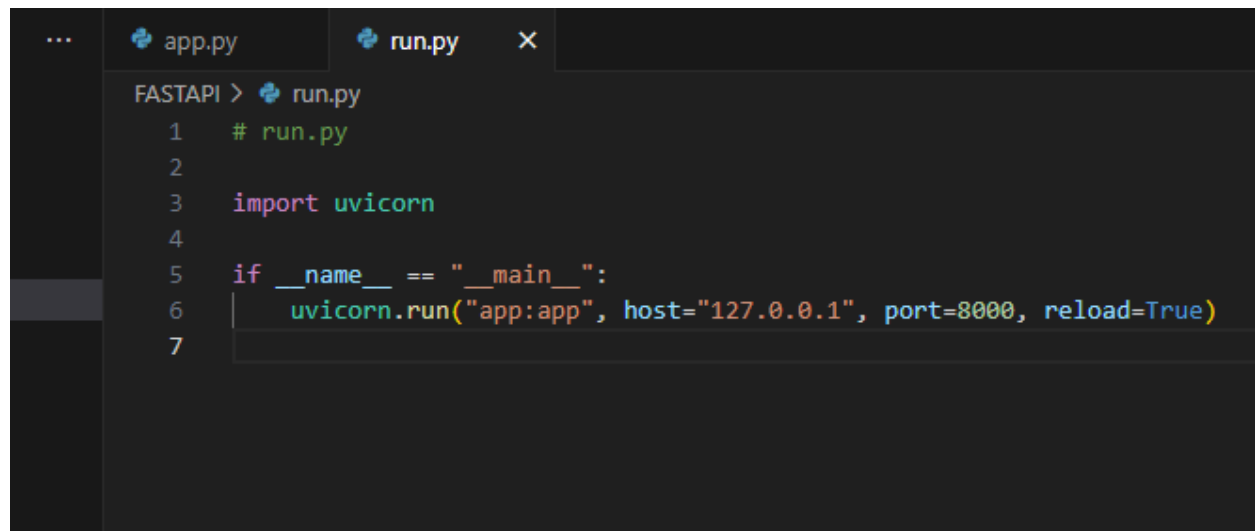
```
def home():
```

```
    # The function associated with the route. It returns a JSON response with a key  
    "Data" and value "Test".
```

```
    return {"Data": "Test"} # Return a simple JSON response
```

Running Your FastAPI Server

Create another file named `run.py` to run the server:

A screenshot of a code editor with a dark theme. The editor has two tabs at the top: 'app.py' and 'run.py'. The 'run.py' tab is active. The code in 'run.py' is as follows:

```
FASTAPI > run.py
1  # run.py
2
3  import uvicorn
4
5  if __name__ == "__main__":
6      uvicorn.run("app:app", host="127.0.0.1", port=8000, reload=True)
7
```

Run the server with the command: `python run.py`

Open your browser and go to the address displayed on your command line, for example: `http://127.0.0.1:8000/`. You should see the following output:

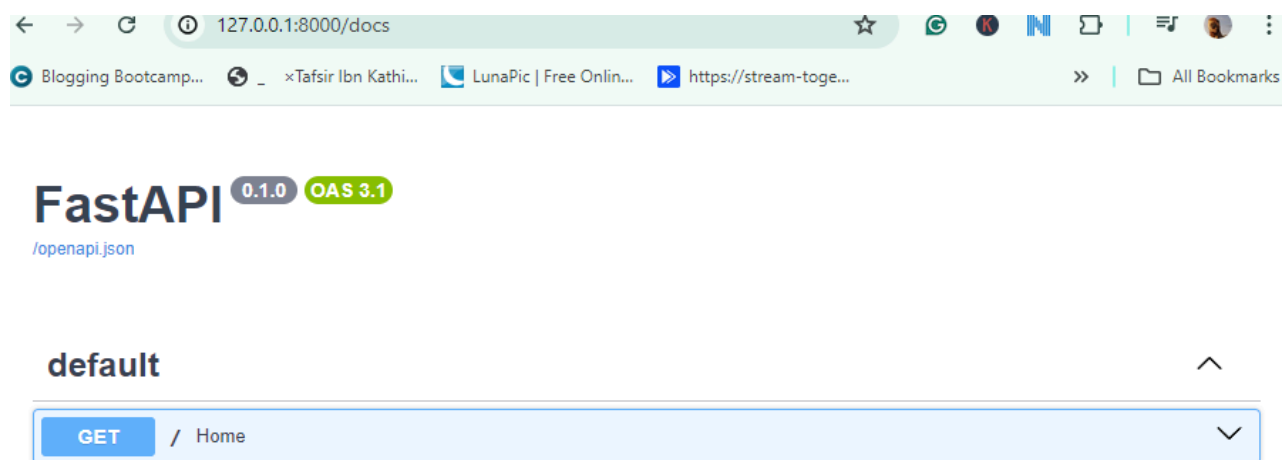

```
{  
  "Data": "Test"  
}
```

Path parameter

Viewing API Documentation with Swagger UI

Swagger UI is an interactive documentation tool for APIs that allows users to explore and test endpoints, and it integrates seamlessly with FastAPI to automatically generate and serve API documentation based on the OpenAPI Specification defined in the FastAPI application.

When you add `/docs` to the end of your HTTP link, it will show you the documentation of your API.



Add the following code to your `app.py` file:

```
@app.get("/about")
def about():
    return {"Data": "About"}
```

In your documentation you should see that an about was added to it:

default		^
GET	/ Home	v
GET	/about About	v

Posting request using Postman

To post a request using Postman, start by downloading Postman from this link: [Postman Downloads](<https://www.postman.com/downloads/>).

Next, type the following code inside the file you created, omitting the comments:

```
from fastapi import FastAPI # Importing the FastAPI class from the fastapi module to
create a new FastAPI app.

from pydantic import BaseModel # Importing the BaseModel class from the pydantic
module to create data models for validation.


app = FastAPI() # Creating an instance of the FastAPI class, which will be our main
application.


class Burrito(BaseModel): # Defining a Pydantic model named Burrito, which describes
the structure of the data.

    name: str # The Burrito model has a 'name' field that should be a string.
    ingredients: list # The Burrito model has an 'ingredients' field that should be a list.
    price: float # The Burrito model has a 'price' field that should be a float.


@app.post("/make_burrito/") # Defining a POST endpoint at the path
"/make_burrito/". This function will be called when a POST request is made to this path.
async def make_burrito(burrito: Burrito): # This asynchronous function will handle the
POST request, taking a Burrito model as input.

    return {"message": f"Your {burrito.name} with {burrito.ingredients} is ready!", "price":
burrito.price} # Returning a JSON response with a message and the price of the burrito.
```

In Postman, add `/make_burrito` to your link, select 'raw' and 'json', and type the following:

```
{ "name": "Veggie Burrito", "ingredients": ["beans", "rice", "cheese", "guac", "salsa"], "price": 7.99 }
```

Then press send.



you will receive a response like this:

```
{  
  "message": "Your Veggie Burrito with ['beans', 'rice', 'cheese', 'guac', 'salsa'] is ready!",  
  "price": 7.99  
}
```

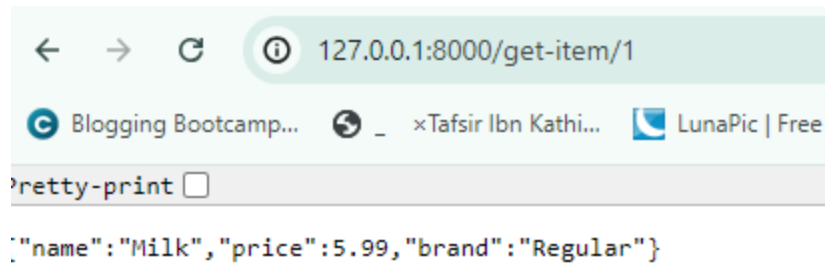
```
}
```

Now, let's create a dictionary to store items and retrieve values via the browser.
Type the following:

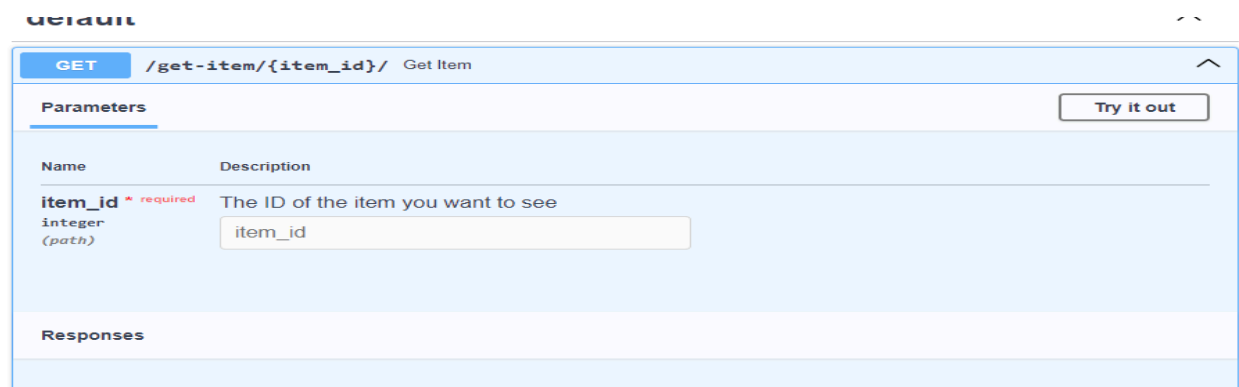
```
inventory = { # Creating a dictionary named 'inventory' to store items with their
              details.
    1: { # The key '1' represents the item ID.
        "name": "Milk", # The item has a 'name' field with the value "Milk".
        "price": 5.99, # The item has a 'price' field with the value 5.99.
        "brand": "Regular" # The item has a 'brand' field with the value "Regular".
    }
}

@app.get("/get-item/{item_id}") # Defining a GET endpoint at the path
"/get-item/{item_id}". This function will be called when a GET request is made to this
path with an item ID.
def get_item(item_id: int): # This function will handle the GET request, taking 'item_id'
as a path parameter and converting it to an integer.
    return inventory[item_id] # Returning the details of the item from the 'inventory'
dictionary corresponding to the provided 'item_id'.
```

In your browser, add `/get-item/1` to your link, keeping the server running, and you will see the following output.



Additionally, if you navigate to `/docs`, you will see the automatically generated API documentation.



CRUD OPERATIONS USING FASTAPI

Query parameter:

Imagine you're at an ice cream parlor. The main flavor you choose is like your main URL path, but sometimes you want to add some extra toppings like sprinkles, chocolate chips, or caramel sauce. These extra goodies are like query parameters!

In FastAPI, query parameters are the little extras you add to your URL to customize your request. Here's an example URL with query parameters for an ice cream order:

<https://icecreamparlor.com/order?flavor=chocolate&toppings=sprinkles&size=large>

Add the following code to your `app.py`:

```
@app.get("/get-by-name/") # Defining a GET endpoint at the path "/get-by-name/".
This function will be called when a GET request is made to this path.
def get_item(name: str): # The function takes a query parameter 'name' of type string.
    for item_id in inventory: # Iterating through each item ID in the inventory.
        if inventory[item_id]["name"] == name: # Checking if the item's name matches
the provided name.
            return inventory[item_id] # Returning the item details if a match is found.
    return {"Data": "Not Found"} # Returning a message indicating that the item was not
found if no match is found.
```

Once your server is running, append the following to your URL:
get-by-name/?name=Milk. You will observe the output:

Pretty-print ☐

```
{"name": "Milk", "price": 5.99, "brand": "Regular"}
```

However, if you try any other product name, such as `get-by-name/?name=Bread`, it will not appear, as it is not in the inventory.

To make the query parameter optional, you can use `None` in your function definition:

```
def get_item(name: str = None): # using None make the query parameter not required anymore
```

Alternatively, you can import `Optional` from the typing module:

```
From typing import Optional  
def get_item(name: Optional[str] = None)
```

Request Body and Post Method

Imagine you're ordering something online. You need to send all your details—what you want, your address, and any special instructions—so the delivery person knows exactly what to bring and where to drop it off. This bundle of details you send is like the Request Body in FastAPI. The POST method is like hitting the "Place Order" button on your favorite online store. It tells the server, "Hey, I've got some new data for you!" and sends your delivery package (Request Body) to the server.

To demonstrate this, you'll create a new POST endpoint by typing the following:

```
@app.post("/create-item/{item_id}") # Defining a GET endpoint at the path
"/create-item/{item_id}". This function will be called when a GET request is made to
this path.

def create_item(item_id: int, item: Item):
    if item_id in inventory: # Checking if the item_id already exists in the inventory
dictionary.
        return {"Error": "Item ID already exists."} # Returning an error message if the
item_id already exists.

    # If the item_id doesn't exist, add the new item to the inventory dictionary.
    inventory[item_id] = {"name": item.name, "brand": item.brand, "price": item.price}

    return inventory[item_id] # Returning the details of the newly created item from the
inventory dictionary.
```

Run the server and navigate to the docs. You will see that the create-item path was created. Click on it to expand, then click on "Try it out." Change the item_id to 3, adjust the values as you want, and click "Execute."

POST

/create-item/{item_id}

Create Item

⌵

Parameters

Cancel

Reset

Name	Description
item_id <small>* required</small>	
integer	
(path)	2

Request body required

application/json

```
{
  "name": "CoconutBiscuit",
  "price": 1.5,
  "brand": "oreo"
}
```

1

Execute

Clear

When you scroll down, you will see the following output.

Request URL

http://127.0.0.1:8000/create-item/2

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "name": "CoconutBiscuit", "brand": "oreo", "price": 1.5 }</pre></div><div><div>⌵</div><div>Download</div></div></div> <div><div>Response headers</div><div><pre>content-length: 52 content-type: application/json date: Sat, 13 Jul 2024 17:49:01 GMT server: uvicorn</pre></div></div>

Responses

Code	Description	Links
200	<div><div>Successful Response</div><div>Media type</div><div>application/json</div><div>Controls Accept header.</div><div>Example Value Schema</div><div><pre>"string"</pre></div></div>	No links

Now, add three different items of your choice. Then go to the GET method on your FastAPI, click on "Try it out," and enter the ID you want to retrieve. Do the same with finding by name.

GET

/get-item/{item_id}/

Get Item

^

Parameters

Cancel

Name	Description
item_id * required	The ID of the item you want to see
integer (path)	<input type="text" value="2"/>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
'http://127.0.0.1:8000/get-item/2/' \
-H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/get-item/2/
```

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "name": "Bread", "price": 15.2, "brand": "ricola" }</pre></div><div><div>Download</div></div></div> <div><div>Response headers</div><div><pre>content-length: 46 content-type: application/json date: Sat,13 Jul 2024 21:42:23 GMT server: uvicorn</pre></div></div>

GET

/get-by-name/Get Item

Parameters

Cancel

Name	Description
name	Name of item.
string (query)	<div>Bread</div>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/get-by-name/?name=Bread' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/get-by-name/?name=Bread
```

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "name": "Bread", "price": 15.2, "brand": "ricola" }</pre></div><div><div></div>Download</div></div> <div><div>Response headers</div><div><pre>content-length: 46 content-type: application/json date: Sat,13 Jul 2024 21:44:56 GMT server: uvicorn</pre></div></div>

Updating an item

To update an item, type the following code:

```
@app.put("/update-item/{item_id}") # Defining a PUT endpoint that takes an item_id
as a path parameter.

def update_item(item_id: int, item: UpdateItem): # Defining the update_item function
which takes item_id as an integer and item of type UpdateItem.

    if item_id not in inventory: # Checking if the provided item_id exists in the inventory
dictionary.

        return {"Error": "Item ID does not exist."} # Returning an error message if the
item_id does not exist.

    # If the item_id exists, proceed to update the item in the inventory dictionary.

    if item.name is not None: # Check if the 'name' field of the item is not None.

        inventory[item_id].name = item.name # If not None, update the 'name' field of the
item in the inventory.

    if item.price is not None: # Check if the 'price' field of the item is not None.

        inventory[item_id].price = item.price # If not None, update the 'price' field of the
item in the inventory.

    if item.brand is not None: # Check if the 'brand' field of the item is not None.

        inventory[item_id].brand = item.brand # If not None, update the 'brand' field of the
item in the inventory.

    return inventory[item_id] # Returning the updated item details from the inventory
dictionary.
```

After saving the code, you should see the PUT method on your FastAPI docs. Add a new item with a new ID and set its name and price. Then, go to the PUT method and try to update an item. Finally, go to the GET method to fetch it.

POST

/create-item/{item_id} Create Item

^

Parameters

Cancel

Reset

Name	Description
<div><div>item_id</div><div>* required</div></div> <div>integer</div> <div>(path)</div>	<div>3</div>

Request body required

application/json

⌵

```
{
  "name": "Chocolate",
  "price": 2.00,
  "brand": "noName"
}
```

Execute

Clear

When you scroll down, you will see that the item was updated.

PUT

/update-item/{item_id} Update Item

Cancel

Reset

Parameters

Name	Description
item_id * required integer (path)	<input type="text" value="3"/>

Request body required

application/json

```
{  "name": "cream"}
```

Execute

Clear

When you scroll down you will see that the item was updated

Server response

Code

Details

200

Response body

```
{  "name": "cream",  "price": 2,  "brand": "noName"}
```

Download

Response headers

```
content-length: 45content-type: application/jsondate: Sat, 13 Jul 2024 22:13:00 GMTserver: uvicorn
```


Deleting an item

To delete an item, type the following code:

```
# Deleting an item
@app.delete("/delete-item") # Defining a DELETE endpoint.
def delete_item(item_id: int = Query(..., description="The ID of the item to delete",
gt=0)): # Defining the delete_item function with a required query parameter.
    if item_id not in inventory: # Checking if the provided item_id exists in the inventory
dictionary.
        return {"Error": "ID does not exist"} # Returning an error message if the item_id does
not exist in the inventory.
    del inventory[item_id] # Deleting the item with the specified item_id from the
inventory dictionary.
    return {"Success": "Item deleted"} # Returning a success message indicating the
item was deleted.
```

Just as you did previously you can also delete by Id inside your FASTAPI docs.

Status Codes

Status codes are like stamps from the internet's post office, telling you the fate of your request. Here's a brief explanation of common status codes:

- 200 OK: 🎉 The letter reached its destination! Everything went perfectly.
- 201 Created: 🏆 You sent a package (like a new account or a blog post), and it got delivered successfully. New stuff has been created!
- 204 No Content: 😊 Your letter was received, but there's no need for a response. Thanks, but no words necessary.
- 400 Bad Request: 😞 Uh-oh! Your letter had gibberish. The post office can't understand what you want.
- 401 Unauthorized: 🚫 The post office needs to see your ID. You can't send letters without proving who you are.
- 403 Forbidden: 🙅 Even with your ID, you're not allowed to send this letter. Some things are just off-limits.
- 404 Not Found: 🕵 The address you sent the letter to doesn't exist. Check the address and try again.
- 500 Internal Server Error: 💥 The post office caught on fire! Something went wrong on their end.

In FastAPI, you can create custom error responses by importing `HTTPException` and raising a Python exception.

```
from fastapi import FastAPI, Path, Query, HTTPException, status
```

So far you've been returning the following way:

```
return {"Data": "Not Found"}
```

Now you will raise the exception:

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

Think of status codes as stamps from the internet's post office, and error responses are like detailed notes explaining what went wrong. FastAPI makes it easy to handle both, and you can even add a bit of humor to keep things light! 😊📬

Afterwords

Well done! You've finished the book, and you're awesome. You've learned how to use FastAPI to create APIs, including handling request bodies and using the POST method to submit data. You've practiced creating, updating, and deleting items using CRUD operations, and you've worked with JSON to structure and validate your data. You've also explored using Postman to test your API endpoints and understood the importance of status codes and error responses to manage server communication effectively. Keep coding, keep practicing, and continue building on these skills to become a proficient developer.

DIVE INTO THE WORLD OF FASTAPI WITH THIS COMPREHENSIVE GUIDE THAT TAKES YOU FROM THE FUNDAMENTALS TO ADVANCED TECHNIQUES. LEARN HOW TO CREATE POWERFUL APIS, HANDLE REQUEST BODIES, AND EFFECTIVELY USE THE POST METHOD TO SUBMIT DATA. MASTER THE ART OF CRUD OPERATIONS—CREATE, READ, UPDATE, AND DELETE ITEMS—WHILE LEVERAGING JSON FOR DATA STRUCTURING AND VALIDATION. DISCOVER HOW TO TEST YOUR API ENDPOINTS USING POSTMAN AND UNDERSTAND THE CRITICAL ROLE OF STATUS CODES AND ERROR RESPONSES IN SERVER COMMUNICATION. WITH PRACTICAL EXAMPLES AND CLEAR EXPLANATIONS, THIS BOOK EQUIPS YOU WITH THE SKILLS TO BECOME A PROFICIENT FASTAPI DEVELOPER.

MASTERING

 FastAPI

