FASTAPI

BEYOND CRUD

Installation and Project SetUp

In chapter, we install FastAPI, starting with a minimal setup.

1. Virtual Environment Creation

Begin by creating a virtual environment using the built-in Python module `venv`. If you already have Python installed, you might not need to install it separately. However, on Linux, installation may be necessary based on your distribution. In your commandline or terminal, type the following command.

```
python3 -m venv env
```

This command generates a virtual environment in the specified folder (in our example, `env`). This folder is an isolated Python environment which shall separate the dependencies of our project from the system wide Python installation.

Activate the virtual environment using the following commands:

On Linux or macOS:

```
source env/bin/activate
```

On Windows:

```
env/Scripts/activate
```

Once activated, your command line will indicate the active virtual environment:

On Linux or macOS:

```
(env) yourusername@yourmachine$
```

On Windows:

```
(env) C:\users\YourUsername>
```

2. Directory Structure

At this point, your directory structure should look like this:

```
└── env
```

3. Installing FastAPI

Now, let us install FastAPI within the virtual environment. We shall install FastAPI with the `pip` using the following command.

```
(env) pip install fastapi
```

4. Freeze Dependencies

Freeze the installed dependencies into a `requirements.txt` file to track the exact versions of our dependencies so that we can easily reproduce them in the future.

```
(env) pip freeze > requirements.txt
```

5. Confirm the installation

Let us confirm our fastapi installation by running the following command.

```
(env) fastapi --version
FastAPI CLI version: 0.0.2
```

This command shall show us the version of FastAPI CLI the FastAPI commandline interface. The `fastapi` command shall help us run and manage our app as we shall see in the coming chapters.

If the version has been displayed, then we are sure that FastAPI has been installed in our virtual environment.

Conclusion

By following these steps, you have successfully set up a virtual environment, and installed FastAPI, frozen the dependencies for future reproducing of the project and you have verified your installation using the `fastapi` CLI command. This structured approach ensures a clean and manageable development environment for our FastAPI project. Next, we shall create a simple web server and create our first API routes.

Creating a Simple Web Server

Introduction

Now that we have FastAPI installed, we are going to create a simple web server on whicn our application shall run using FastAPI.

At this stage, our directory only contains our virtual environment directory `env` and `requirements.txt` as shown below as follows:

Current directory structure

```
└── env
└── requirements.txt
```

Let's create a file named `main.py` and populate it with the following code:

main.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get('/')
async def read_root():
    return {"message": "Hello World!"}
```

In this code snippet, we perform the following actions:

Creating a FastAPI instance:

We have imported the `FastAPI` class from the `fastapi` package. This class serves as the primary entry point for all FastAPI applications. Through it we can get access to various FastAPI features such as routes , middleware, exception handlers and path operations.

We then create an instance of the `FastAPI` class named `app`. The main FastAPI instance can be called anything as long as it is a valid Python name.

Creating the FastAPI instance

```
from fastapi import FastAPI

app = FastAPI()
```

Creating an API Route:

We define our first API route by creating a function named `read_root`. This function, when accessed, will return a JSON message containing "Hello World!".

Your first API endpoint

```
@app.get('/')
async def read_root():
    return {"message": "Hello World!"}
```

The `@app` decorator associates the function with the HTTP GET method via the `get` method. We then provide the path (route) of the root path (`/`). This means that whenever the `/` route is accessed, the defined message will be returned.

All HTTP methods such as `post`, `put`, `head`, `patch`, `delete`, `trace` and `options` are all available on the `@app` decorator.

Running the FastAPI Application:

To run our FastAPI application, we shall use the `fastapi` command we introduced in the previous chapter. Open a terminal and execute the following command within the virtual environment:

Running the server with the FastAPI CLI

```
(env)$ fastapi dev main.py
```

The `fastapi dev` command enables us to execute our FastAPI application in development mode. This feature facilitates running our application with auto-reload functionality, ensuring that any modifications we make are automatically applied, restarting the server accordingly. It operates by identifying the FastAPI instance within the specified module or Python package, which in our scenario is `main.py`, where we have defined the app object. When we initiate our application, it will display the following output.

terminal ouput

```
INFO     Using path main.py
INFO     Resolved absolute path /home/jod35/Documents/fastapi-beyond-CRUD/main.py
INFO     Searching for package file structure from directories with __init__.py files
INFO     Importing from /home/jod35/Documents/fastapi-beyond-CRUD

 ╭─ Python package file structure ─╮
 │                                 │
 │      🐍 main.py                 │
 │                                 │
 ╰─────────────────────────────────╯

INFO     Importing module main.py
INFO     Found importable FastAPI app

 ╭─ Importable FastAPI app ─╮
 │                          │
 │  from main import app    │
 │                          │
 ╰──────────────────────────╯

INFO     Using import string main:app
```

```
╭─────────── FastAPI CLI - Development mode ────────────╮
│                                                       │
│  Serving at: http://127.0.0.1:8000                    │
│                                                       │
│  API docs: http://127.0.0.1:8000/docs                 │
│                                                       │
│  Running in development mode, for production use:     │
│                                                       │
│  fastapi run                                          │
│                                                       │
```

Running the server will make your application available at this web address: `http://localhost:8000`.

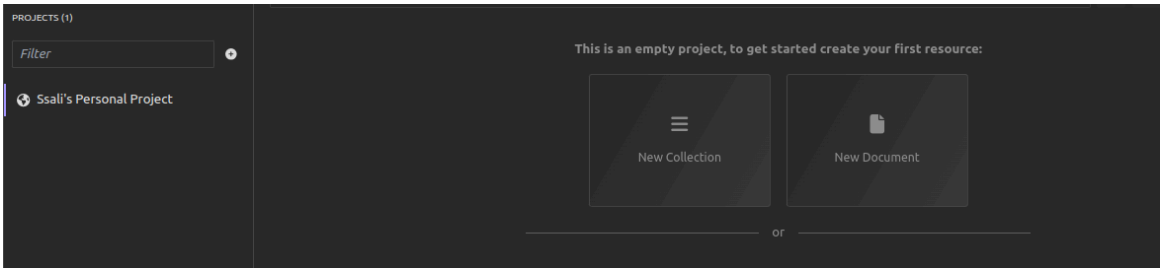
Following these steps means you've successfully created a basic FastAPI application with a greeting endpoint. You've also learned how to start the application using Uvicorn, which helps you develop more easily because it automatically reloads your code when you make changes.

Choosing an API Client

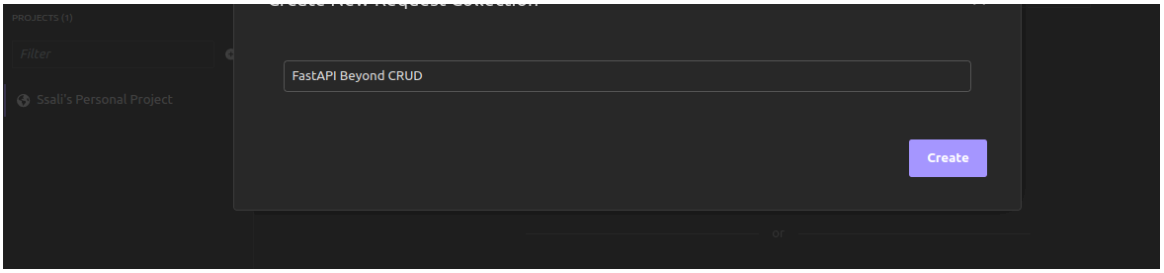
Depending on your choice, you may want to test your application with an Api Client, I will begin with Insomnia which is a simple open source application for testing and development APIs.

In insomnia, we shall create our simple request collection and we shall now see our response of `Hello World`.

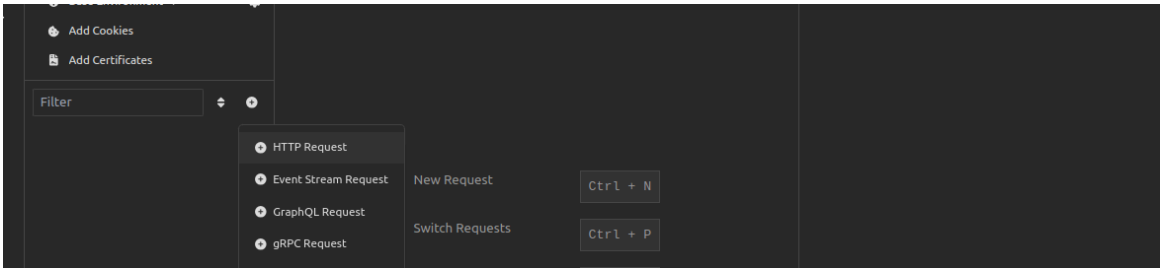
1. Create a new request collection



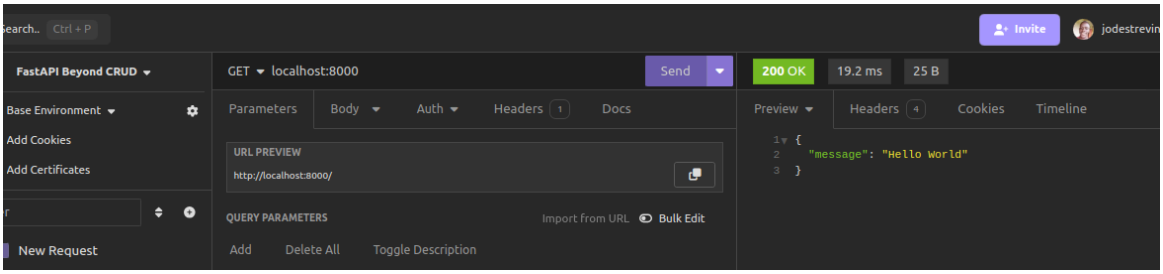
2. Name the request collection



3. Create an HTTP request



4. Make a request



And just like that, you have created your FastAPI application, run it and even made your HTTP request using an HTTP client.

Managing Requests and Responses

There are very many ways that clients can pass request data to a FastAPI API route. These include: - Path Parameters - Query Parameters - Headers e.t.c.

Through such ways, we can obtain data from incoming requests to our APIs.

Parameter type declarations

All parameters in a FastAPI request are requiresd to have a type declaration via *type hints*. Primitive Python types such (`None` , `int` , `str` , `bool` , `float`), container types such as (`dict` , `tuples` , `dict` , `set`) and some other complex types are all supported.

Additionally FastAPI also allows all types present within Python's `typing` module. These data types represent common conventions in Python and are utilized for variable type annotations. They facilitate type checking and model validation during compilation. Examples include `Optional` , `List` , `Dict` , `Set` , `Union` , `Tuple` , `FrozenSet` , `Iterable` , and `Deque` .

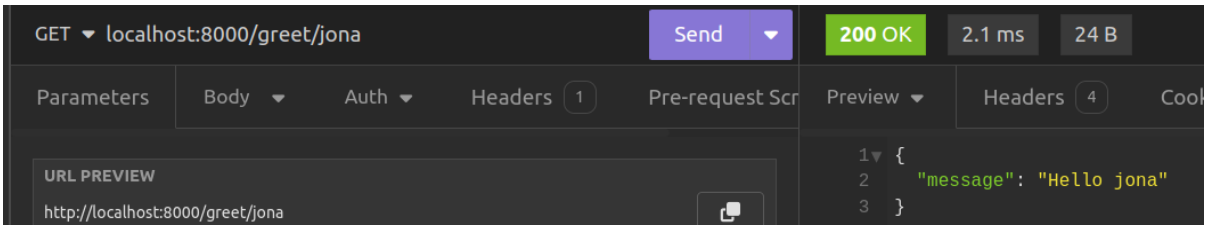
Path Parameters

All request data supplied in the endpoint URL of a FastAPI API is acquired through a path parameter, thus rendering URLs dynamic. FastAPI adopts curly braces (`{}`) to denote path parameters when defining a URL. Once enclosed within the braces, FastAPI requires that they be provided as parameters to the route handler functions we establish for those paths.

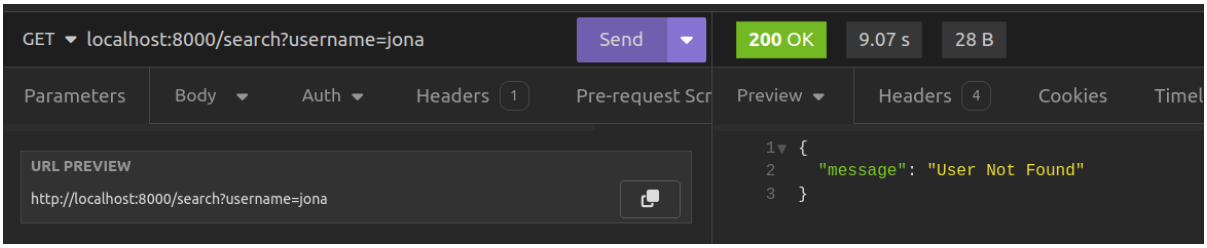
path parameters

```
#inside main.py
@app.get('/greet/{username}')
async def greet(username:str):
   return {"message":f"Hello {username}"}
```

In this example the `greet()` route handler function will require `username` which is annotated with `str` indicating that the username shall be a string. Sending a greetings to the user "jona" shall return the response shown below.



Just in we make a request to the route without the param,



Query Parameters

These are key-value pairs provided at the end of a URL, indicated by a question mark (`?`). Just like path parameters, they also take in request data. Whenever we want to provide multiple query parameters, we use the ampersand (`&`) sign.

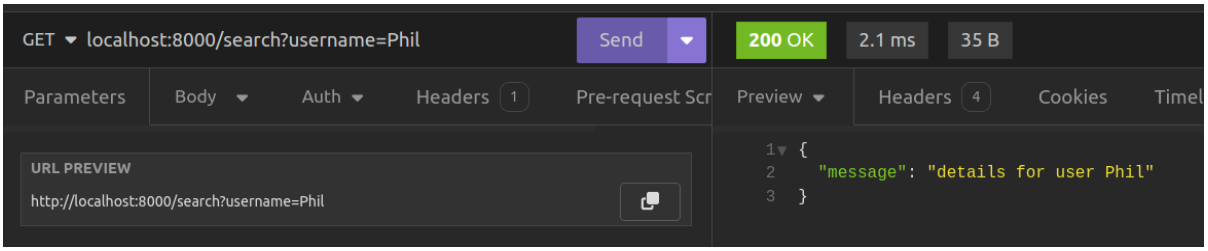
Query params

```
# inside main.py

user_list = [
   "Jerry",
   "Joey",
   "Phil"
]

@app.get('/search')
async def search_for_user(username:str):
   for user in user_list:
    if username in user_list :
        return {"message":f"details for user {username}"}

    else:
        return {"message":"User Not Found"}
```

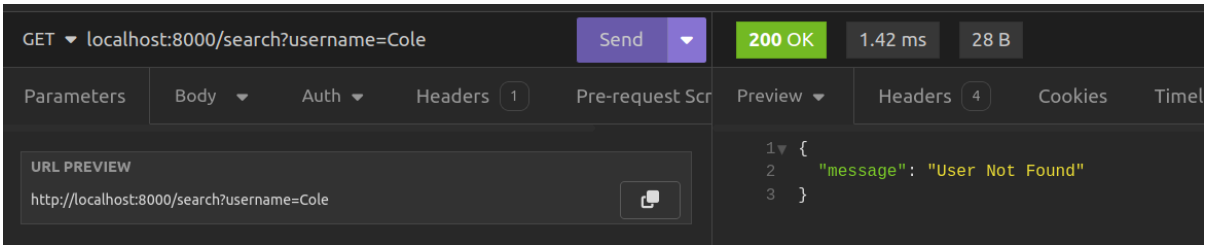
In this example, we've set up a route for searching users within a simple list. Notice that there are no path parameters specified in the route's URL. Instead, we're passing the `username` directly to our route handler, `search_for_user`. In

FastAPI, any parameter passed to a route handler, like `search_for_user`, and is not provided in the path as a path param is treated as a query parameter. Therefore, to access this route, we need to use `/search?username=sample_name`, where `username` is the key and `sample_name` is the value.

Let us save and test the example above. Saecrning for a user who exists returns the needed response.



And searching for a user who does not exist returns the following response.



Optional Parameters

There may also be cases when the API route can operate as needed even in the presence of a path or query param. In this case, we can make the parameters optional when annotating their types in the route handler functions. Forexample, our first example can be modified to the following:

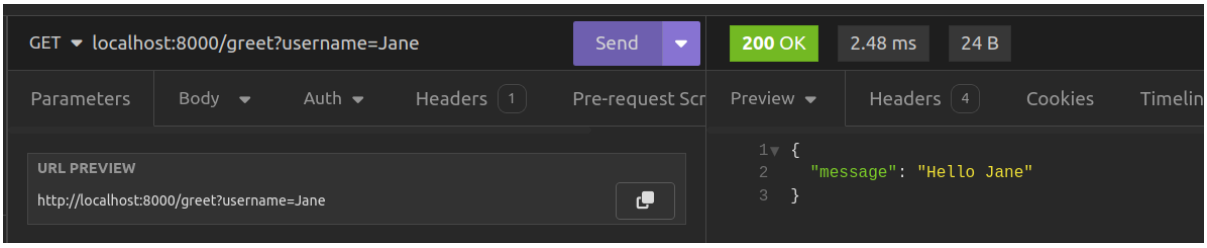
Optional Query Params

```
from typing import Optional

@app.get('/greet/')
async def greet(username:Optional[str]="User"):
   return {"message":f"Hello {username}"}
```

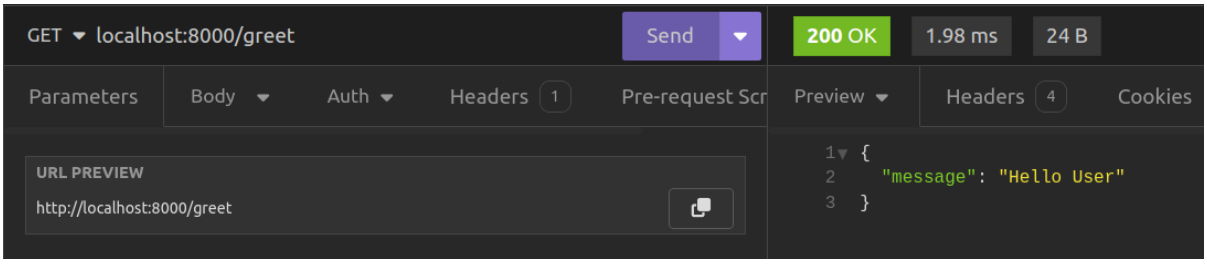
This time, we've made the `username` path parameter optional. We achieved this by removing it from the route definition. Additionally, we updated the type annotation for the `username` parameter in the `greet` route handler function to make it an optional string, with a default value of "User". To accomplish this, we're using the `Optional` type from Python's `typing` module.

```
username:Optional[str]
```

When we save the example, we shall test it and get the following response.



Note that this time if we do not provide the `username`, we shall get the default username of "User".



Request Body

Frequently, clients need to send data to the server for tasks like creating or updating resources through methods like POST, PATCH, PUT, DELETE, or for various other operations. FastAPI simplifies this process by enabling you to define a Pydantic model to establish the structure of the data being sent. Furthermore, it aids in validating data types using type hints. Let's delve into a straightforward example to illustrate this concept.

Request Body

```
# inside main.py
from pydantic import BaseModel

# the User model
class UserSchema(BaseModel):
   username:str
   email:str


@app.post("/create_user")
async def create_user(user_data:UserSchema):
   new_user = {
      "username" : user_data.username,
      "email": user_data.email
   }

   users.append(new_user)

   return {"message":"User Created successfully","user":new_user}
```

What we have done in the above example is to create a Pydantic model by inheriting Pydantic's `BaseModel` class. On this class we have defined attributes `username` and `email` and also annotated them with the `str` type.

A simple Pydantic model

```
class UserSchema(BaseModel):
   username:str
   email:str
```

Following that, have crafted an API route intended to handle a POST request at /create_user. The handler for this route accepts a parameter representing the user_data obtained from the client, with its type annotated as the Pydantic model UserSchema.

```
create_user(user_data:UserSchema)
```

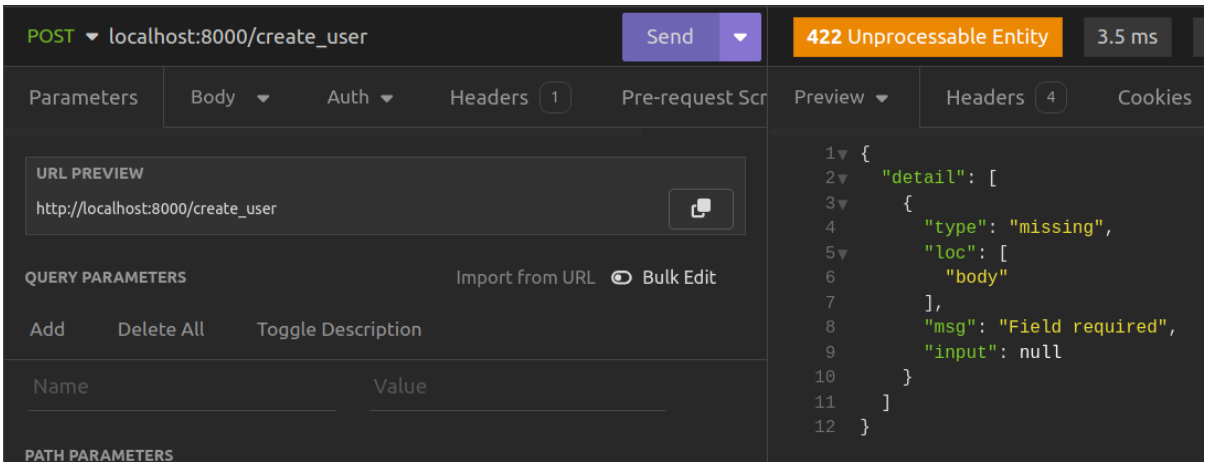
Using this `user_data`, we construct a new_user dictionary and append it to our users list.

```
   new_user = {
      "username" : user_data.username,
      "email": user_data.email
   }

   users.append(new_user)
```

We finally return a response with the newly created `new_user` dictionary.

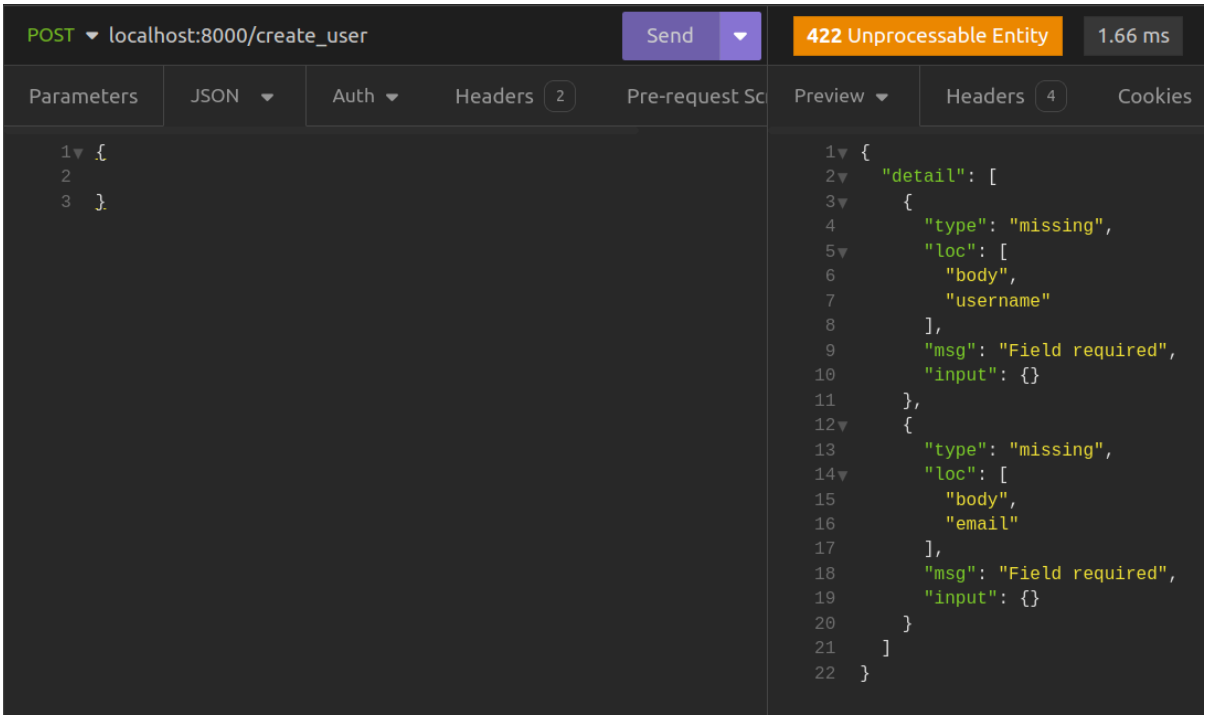
```
   return {"message":"User Created successfully","user":new_user}
```

Let's test this. If we make the request without providing the request body, we should receive the following response.



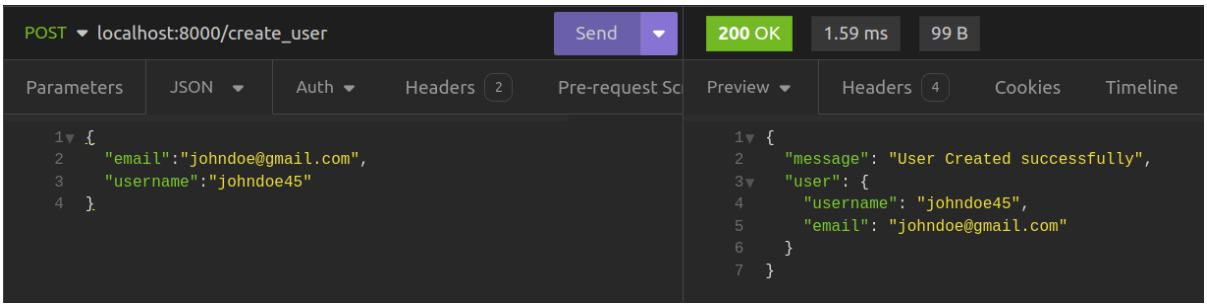
Please note that we will receive the `422 Unprocessable Entity` status code because FastAPI failed to retrieve data from the request body, as it has not been provided.

When we provide the body without the required fields, we will receive the following output.



We will receive the same output, but this time we will be notified that the required fields (`email` , `username`) for the data are missing.

Let us now provide valid data.



Supplying valid values for the `email` and `username` fields will result in a successful response.

Note

There can indeed be scenarios that necessitate the use of all the features we've discussed. You can have an API route that accepts path, query, and optional parameters, and FastAPI is capable of handling such complexity seamlessly.

Request Headers

During a request-response transaction, the client not only sends parameters to the server but also provides information about the context of the request's origin. This contextual information is crucial as it enables the server to customize the type of response it returns to the client.

Common request headers include: - `User-Agent` : This string allows network protocol peers to identify the application responsible for the request, the operating system it's running on, or the version of the software being used.

- `Host` : This specifies the domain name of the server, and (optionally) the TCP port number on which the server is listening.
- `Accept` : Informs the server about the types of data that can be sent back.
- `Accept-Language` : This header informs the server about the preferred human language for the response.
- `Accept-Encoding` : The encoding algorithm, usually a compression algorithm, that can be used on the resource sent back.
- `Referer` : This specifies the address of the previous web page from which a link to the currently requested page was followed.
- `Connection` : This header controls whether the network connection stays open after the current transaction finishes.

To access such headers, FastAPI provides us with the `Header` function giving us the ability to get the values of these headers using the exact names but in a snake-case syntax forexample, `User-Agent` is `user_agent` , `Accept-Encoding` is `accept_encoding` and so on. Let us take a look at a small code example.

Request Headers

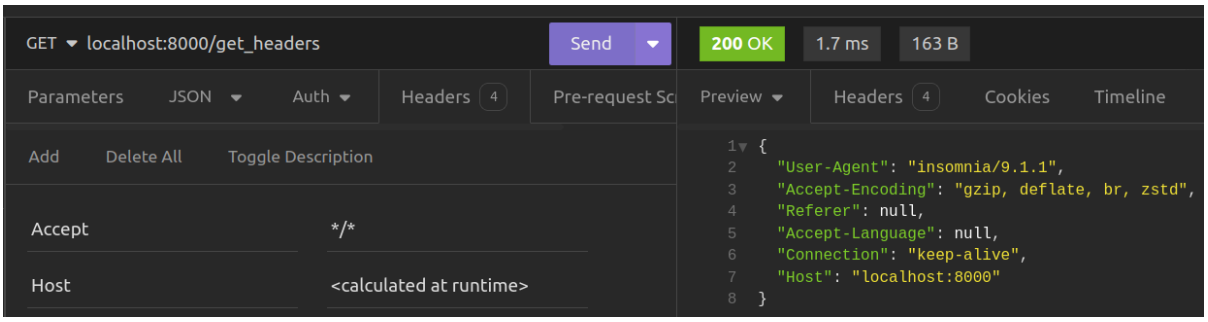
```
# inside main.py
@app.get('/get_headers')
async def get_all_request_headers(
    user_agent: Optional[str] = Header(None),
    accept_encoding: Optional[str] = Header(None),
    referer: Optional[str] = Header(None),
    connection: Optional[str] = Header(None),
    accept_language: Optional[str] = Header(None),
    host: Optional[str] = Header(None),
):
    request_headers = {}
    request_headers["User-Agent"] = user_agent
    request_headers["Accept-Encoding"] = accept_encoding
    request_headers["Referer"] = referer
    request_headers["Accept-Language"] = accept_language
    request_headers["Connection"] = connection
    request_headers["Host"] = host

    return request_headers
```

We've started by importing the `Header` function from FastAPI into our route handler. Each header has been added and designated as an optional string. A default value has been assigned by invoking the `Header` function with `None` as a parameter. Using the `None` argument allows the Header() function to declare the variable optionally, which aligns with best practices.

```
    user_agent: Optional[str] = Header(None)
```

we have then created a request_headers dictionary with the names of the headers as keys and the values as what we get by calling the `Header` function.

Making a request to the `/get_headers` route shall return the following response depending on how you have made the request. For my case,



Conclusion

In this chapter, we utilized FastAPI to construct a basic web server and delved into different methods of communicating and inputting data. We introduced concepts such as path and query parameters, along with request headers. Moving forward, the next chapter will focus on developing a straightforward CRUD API for managing book resources utilizing an in-memory database.

Building a CRUD REST API

What is CRUD?

CRUD represents the four basic data operations:

- **Create (C):**
- *Objective:* Add new data.
- *Action:* Insert a new record or entity.
- **Read (R):**
- *Objective:* Retrieve existing data.
- *Action:* Fetch data without modification.
- **Update (U):**
- *Objective:* Modify existing data.
- *Action:* Update attributes or values.
- **Delete (D):**
- *Objective:* Remove data.
- *Action:* Delete a record or entity.

CRUD operations are fundamental in data management, commonly used in applications dealing with data persistence. In **FastAPI Beyond CRUD**, the focus is on extending FastAPI capabilities beyond typical CRUD applications, exploring advanced features and use cases. But before diving into such aspects, let us build a simple CRUD API using FastAPI.

A simple CRUD API implementation

Our simple CRUD API will have a few endpoints to perform CRUD operations on a simple in-memory database of books. Here's a list of endpoints that we shall have in our CRUD API.

Endpoint	Method	Description
/books	Get	Read all books
/books	POST	Create a book
/book/{book_id}	GET	Get a book by id
/book/{book_id}	PATCH	Update a book by id

Endpoint	Method	Description
/book/{book_id}	DELETE	Delete a book by id

The provided table describes various API endpoints, their associated HTTP methods, and their functionalities:

1. `/books` **- GET: Read all books**
2. *Description:* This endpoint is designed to retrieve information about all available books. When a client makes an HTTP GET request to `/books`, the server responds by providing details on all books in the system.
3. `/books` **- POST: Create a book**
4. *Description:* To add a new book to the system, clients can make an HTTP POST request to `/books`. This operation involves creating and storing a new book based on the data provided in the request body.
5. `/book/{book_id}` **- GET: Get a book by id**
6. *Description:* By making an HTTP GET request to `/book/{book_id}`, clients can retrieve detailed information about a specific book. The `book_id` parameter in the path specifies which book to fetch.
7. `/book/{book_id}` **- PATCH: Update a book by id**
8. *Description:* To modify the information of a specific book, clients can send an HTTP PATCH request to `/book/{book_id}`. The `book_id` parameter identifies the target book, and the request body contains the updated data.
9. `/book/{book_id}` **- DELETE: Delete a book by id**
10. *Description:* This endpoint allows clients to delete a specific book from the system. By sending an HTTP DELETE request to `/book/{book_id}`, the book identified by `book_id` will be removed from the records.

Now that we have a plan of our simple API, we can now build our simple CRUD API by adding the following code to `main.py`. We shall begin by creating a very simple list of books that we will use as our database.

in memory database of the books

```
books = [
    {
        "id": 1,
        "title": "Think Python",
        "author": "Allen B. Downey",
        "publisher": "O'Reilly Media",
        "published_date": "2021-01-01",
        "page_count": 1234,
        "language": "English",
    },
    {
        "id": 2,
        "title": "Django By Example",
        "author": "Antonio Mele",
        "publisher": "Packt Publishing Ltd",
        "published_date": "2022-01-19",
        "page_count": 1023,
        "language": "English",
    },
    {
        "id": 3,
```

```
        "title": "The web socket handbook",
        "author": "Alex Diaconu",
        "publisher": "Xinyu Wang",
        "published_date": "2021-01-01",
        "page_count": 3677,
        "language": "English",
    },
    {
        "id": 4,
        "title": "Head first Javascript",
        "author": "Hellen Smith",
        "publisher": "Oreilly Media",
        "published_date": "2021-01-01",
        "page_count": 540,
        "language": "English",
    },
    {
        "id": 5,
        "title": "Algorithms and Data Structures In Python",
        "author": "Kent Lee",
        "publisher": "Springer, Inc",
        "published_date": "2021-01-01",
        "page_count": 9282,
        "language": "English",
    },
    {
        "id": 6,
        "title": "Head First HTML5 Programming",
        "author": "Eric T Freeman",
        "publisher": "O'Reilly Media",
        "published_date": "2011-21-01",
        "page_count": 3006,
        "language": "English",
    },
]
```

Once we have that, we shall build our endpoints on the simple database.

All CRUD endpoints on the in-memory store

```
from fastapi import FastAPI, status
from fastapi.exceptions import HTTPException
from pydantic import BaseModel
from typing import List


app = FastAPI()

class Book(BaseModel):
    id: int
    title: str
    author: str
    publisher: str
    published_date: str
    page_count: int
    language: str

class BookUpdateModel(BaseModel):
    title: str
    author: str
    publisher: str
```

```
    page_count: int
    language: str


@app.get("/books", response_model=List[Book])
async def get_all_books():
    return books


@app.post("/books", status_code=status.HTTP_201_CREATED)
async def create_a_book(book_data: Book) -> dict:
    new_book = book_data.model_dump()

    books.append(new_book)

    return new_book


@app.get("/book/{book_id}")
async def get_book(book_id: int) -> dict:
    for book in books:
        if book["id"] == book_id:
            return book

    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Book not found")


@app.patch("/book/{book_id}")
async def update_book(book_id: int,book_update_data:BookUpdateModel) -> dict:

    for book in books:
        if book['id'] == book_id:
            book['title'] = book_update_data.title
            book['publisher'] = book_update_data.publisher
            book['page_count'] = book_update_data.page_count
            book['language'] = book_update_data.language

            return book

    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Book not found")


@app.delete("/book/{book_id}",status_code=status.HTTP_204_NO_CONTENT)
async def delete_book(book_id: int):
    for book in books:
        if book["id"] == book_id:
            books.remove(book)

            return {}

    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Book not found")
```

Reading All Books (HTTP GET)

This route responds to GET requests made to `/books`, providing a list of all books available in the application. It ensures that the response adheres to the `List[Book]` model, guaranteeing consistency with the structure defined by the `Book` model.

Read all books

```python
class Book(BaseModel):
    id: int
    title: str
    author: str
    publisher: str
    published_date: str
    page_count: int
    language: str


@app.get("/books", response_model=List[Book])
async def get_all_books():
    return books
```

FastAPI significantly simplifies the process of returning any JSON serializable object as a response.

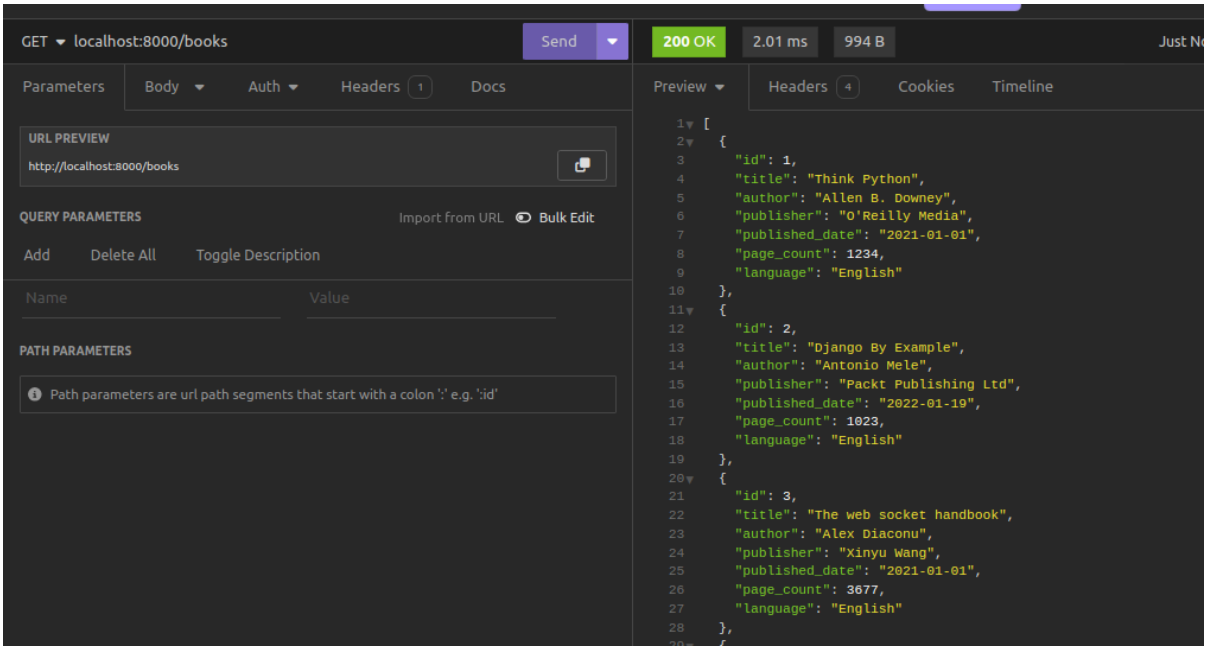
Note

JSON (JavaScript Object Notation) serialization involves transforming a data structure or object from a programming language (such as Python, JavaScript, or others) into a JSON-formatted string. This string representation can then be transmitted over a network or stored in a file, subsequently allowing deserialization back into the original data structure.

In Python, the following data types are serializable:

- Lists
- Dictionaries
- Strings
- Tuples
- Booleans
- None

This capability enables us to effortlessly respond with a list of book objects when issuing a `GET` request to `http://localhost:8000/books`, as illustrated below:



Read one Book (HTTP GET)

To retrieve a single book by its ID, the FastAPI application employs the `read_book` function whenever a request is made to `book/{book_id}`. The `{book_id}` serves as a path parameter passed to the `read_book` function to locate the book with the corresponding ID. The process involves iterating through the list of books to verify the existence of a book with the provided ID. If the book is not found, an `HTTPException` is raised, signaling that the book resource is not available. Notably, FastAPI's `status` module facilitates access to status codes, enabling the use of codes such as `HTTP_404_NOT_FOUND` to indicate resource absence.

Retrieve a book by ID

```
@app.get("/book/{book_id}")
async def get_book(book_id: int) -> dict:
    for book in books:
        if book["id"] == book_id:
            return book

    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Book not found")
```