# Build an App With FastAPI for Python

It's called "fast" for a reason! Here's what you need to know about FastAPI to quickly build application programming interfaces using Python.

By Emmanuel Uchenna

11 min. read · View original

FastAPI is a speedy and lightweight web framework for building modern application programming interfaces using Python 3.6 and above. In this tutorial, we'll walk through the basics of building an app with FastAPI, and you'll get an inkling of why it was nominated as one of the best open-source frameworks of 2021.

Once you're ready to develop your own FastAPI apps, you won't have to look far to find a place to host them. Kinsta's Application Hosting and Database Hosting services provide a Platform as a Service that's strong on Python.

Let's learn the basics first.

## Advantages of FastAPI

Below are some of the advantages the FastAPI framework brings to a project.

- **Speed:** As the name implies, FastAPI is a very fast framework. Its speed is comparable to that of Go and Node.js, which are generally considered to be among the fastest options for building APIs.
- **Easy to learn and code:** FastAPI has already figured out almost everything you will need to make a production-ready API. As a developer using FastAPI, you don't need to code everything from scratch. With only a few lines of code, you can have a RESTful API ready for deployment.
- **Comprehensive documentation:** FastAPI uses the OpenAPI documentation standards, so documentation can be dynamically generated. This documentation provides detailed information about FastAPI's endpoints, responses, parameters, and return codes.
- **APIs with fewer bugs:** FastAPI supports custom data validation, which allows developers to build APIs with fewer bugs. FastAPI's developers boast that the framework results in fewer human-induced bugs — as much as 40% less.
- **Type hints:** The types module was introduced in Python 3.5. This enables you to declare the `type` of a variable. When the type of a variable is declared, IDEs are able to provide better support and predict errors more accurately.

## How to Get Started With FastAPI

To follow this tutorial and get started with FastAPI, you'll need to do a few things first.

Ensure that you have a programmer's text editor/IDE, such as Visual Studio Code. Other options include Sublime Text and Espresso.

It's a common practice to have your Python apps and their instances running in virtual environments. Virtual environments allow different package sets and configurations to run simultaneously, and avoid conflicts due to incompatible package versions.

To create a virtual environment, open your terminal and run this command:

```
$ python3 -m venv env
```

You'll also need to activate the virtual environment. The command to do that will vary depending on the operating system and shell that you're using. Here are some CLI activation examples for a number of environments:

```
# On Unix or MacOS (bash shell):
/path/to/venv/bin/activate

# On Unix or MacOS (csh shell):
/path/to/venv/bin/activate.csh

# On Unix or MacOS (fish shell):
/path/to/venv/bin/activate.fish

# On Windows (command prompt):
\path\to\venv\Scripts\activate.bat

# On Windows (PowerShell):
\path\to\venv\Scripts\Activate.ps1
```

(Some Python-aware IDEs can also be configured to activate the current virtual environment.)

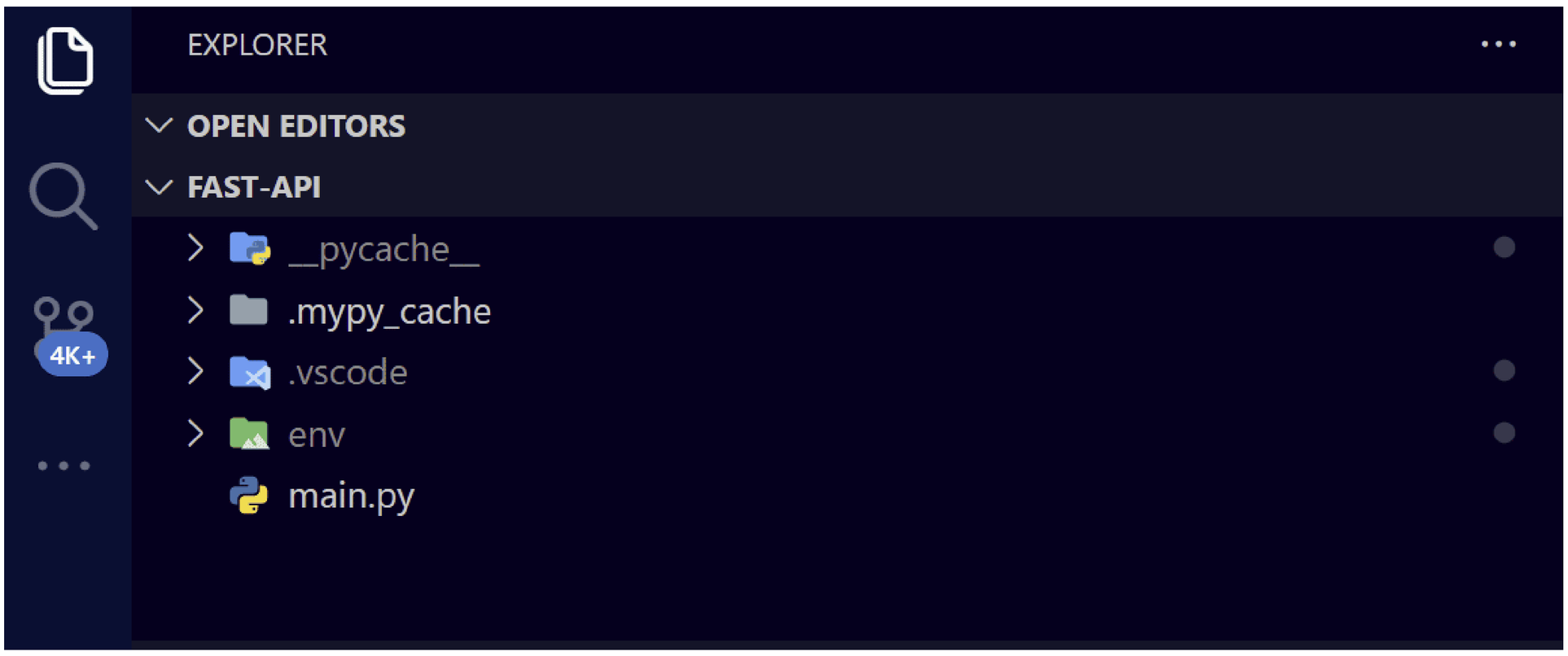
Now, install FastAPI:

```
$ pip3 install fastapi
```

FastAPI is a framework for building APIs, but to test your APIs you'll need a local web server. Uvicorn is a lightning-fast Asynchronous Server Gateway Interface (ASGI) web server for Python that is great for development. To install Uvicorn, run this command:

```
$ pip3 install "uvicorn[standard]"
```

Upon successful installation, create a file named **main.py** within your project's working directory. This file will be your application entry point.




View of a basic FastAPI project within an IDE.

### A Quick FastAPI Example

You will test your FastAPI installation by quickly setting up an example endpoint. In your **main.py** file, paste in the following code, then save the file:

```
# main.py
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def root():
 return {"greeting":"Hello world"}
```

The above snippet creates a basic FastAPI endpoint. Below is a summary of what each line does:

- `from fastapi import FastAPI`: The functionality for your API is provided by the FastAPI Python class.
- `app = FastAPI()`: This creates a FastAPI instance.
- `@app.get("/")`: This is a python decorator that specifies to FastAPI that the function below it is in charge of request handling.
- `@app.get("/")`: This is a decorator that specifies the route. This creates a `GET` method on the site's route. The result is then returned by the wrapped function.
- Other possible operations that are used to communicate include `@app.post()`, `@app.put()`, `@app.delete()`, `@app.options()`, `@app.head()`, `@app.patch()`, and `@app.trace()`.

In the files directory, run the following command in your terminal to start the API server:

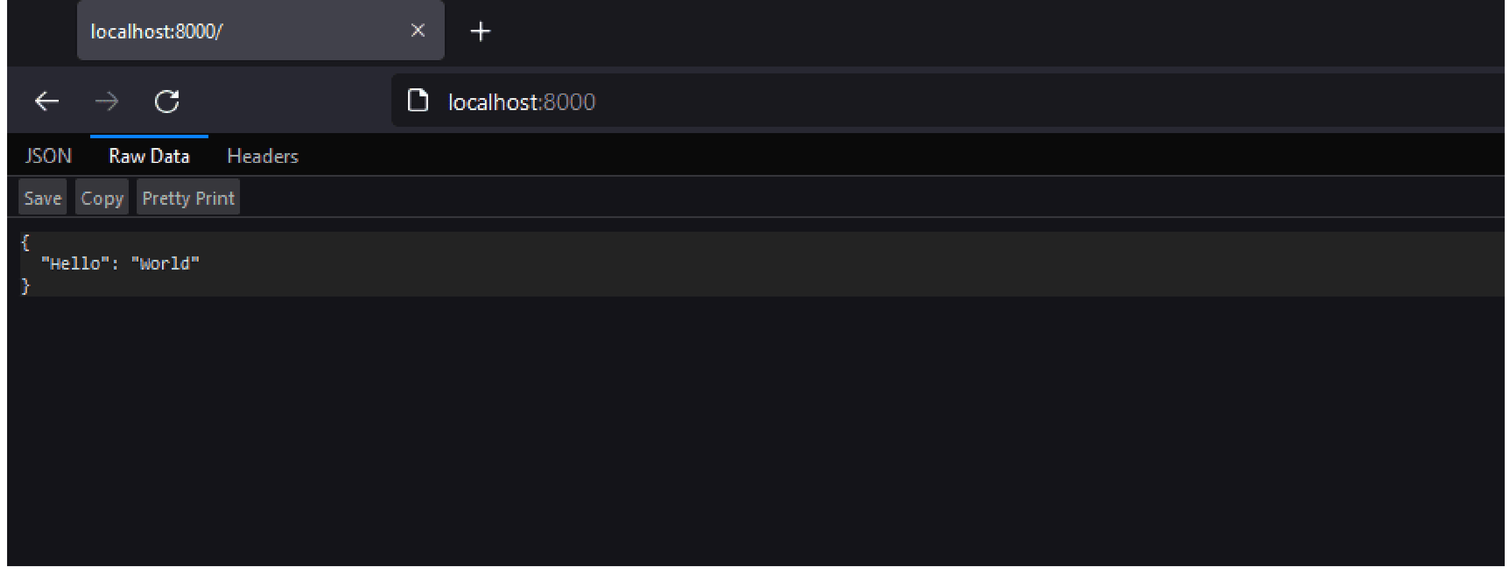
```
$ uvicorn main:app --reload
```

In this command, `main` is the name of your module. The app object is an instance of your application, and is imported into the ASGI server. The `--reload` flag tells the server to reload automatically when you make any changes.

You should see something like this in your terminal:

```
 $ uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['D:\\WEB DEV\\Eunit\\Tests\\fast-api']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [26888] using WatchFiles
INFO: Started server process [14956]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

In your browser, navigate to `http://localhost:8000` to confirm that your API is working. You should see "Hello": "World" as a JSON object on the page. This illustrates how easy it is to create an API with FastAPI. All you had to do was to define a route and return your Python dictionary, as seen on line six of the snippet above.




FastAPI Hello World application in a web browser.

### Using Type Hints

If you use Python, you are used to annotating variables with basic data types such as `int`, `str`, `float`, and `bool`. However, from Python version 3.9, advanced data structures were introduced. This allows you to work with data structures such as `dictionaries`, `tuples`, and `lists`. With FastAPI's type hints, you can structure the schema of your data using pydantic models and then, use the pydantic models to type hint and benefit from the data validation that is provided.

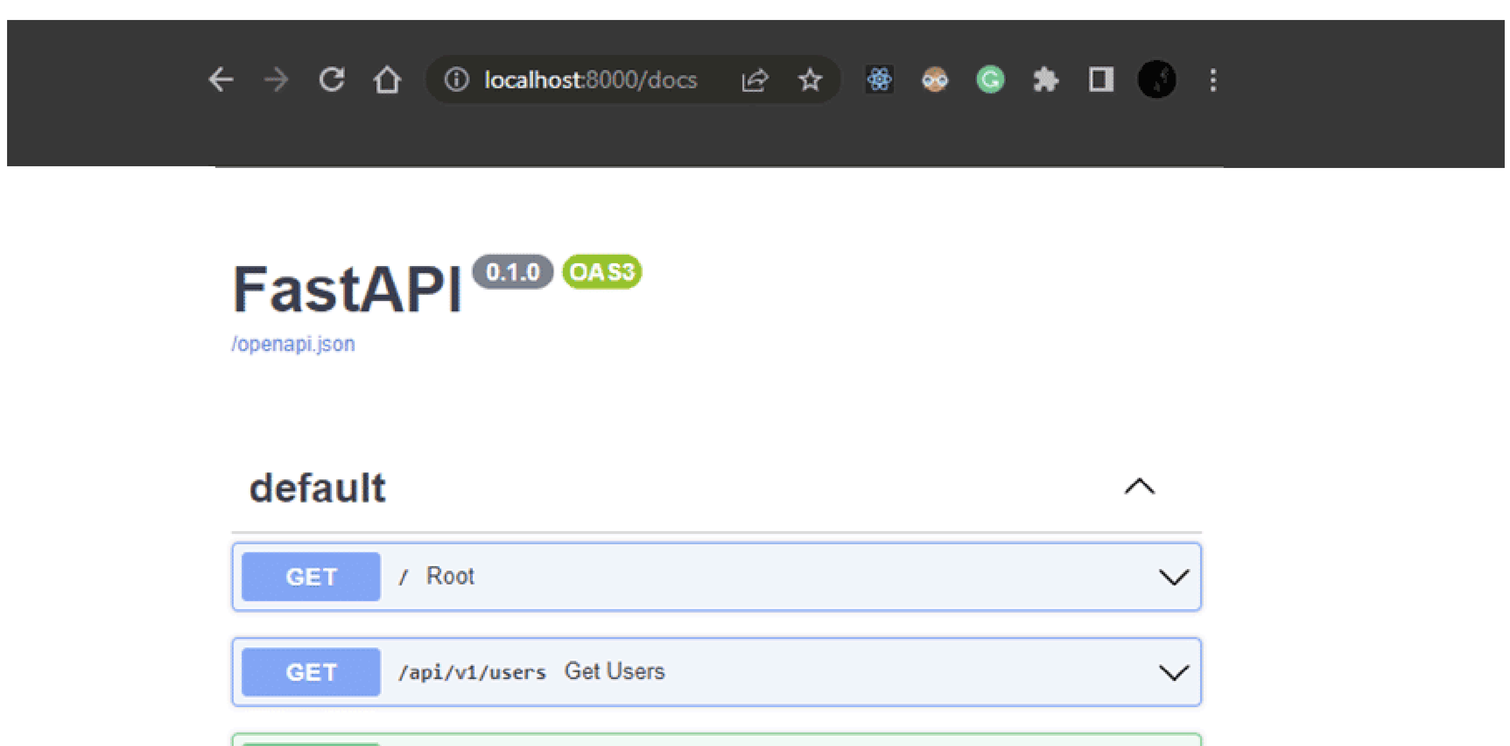
In the example below, the use of type hints in Python is demonstrated with a simple meal price calculator, `calculate_meal_fee`:

```
def calculate_meal_fee(beef_price: int, meal_price: int) -> int:
 total_price: int = beef_price + meal_price
 return total_price
print("Calculated meal fee", calculate_meal_fee(75, 19))
```

Note that type hints do not change how your code runs.

### FastAPI Interactive API Documentation

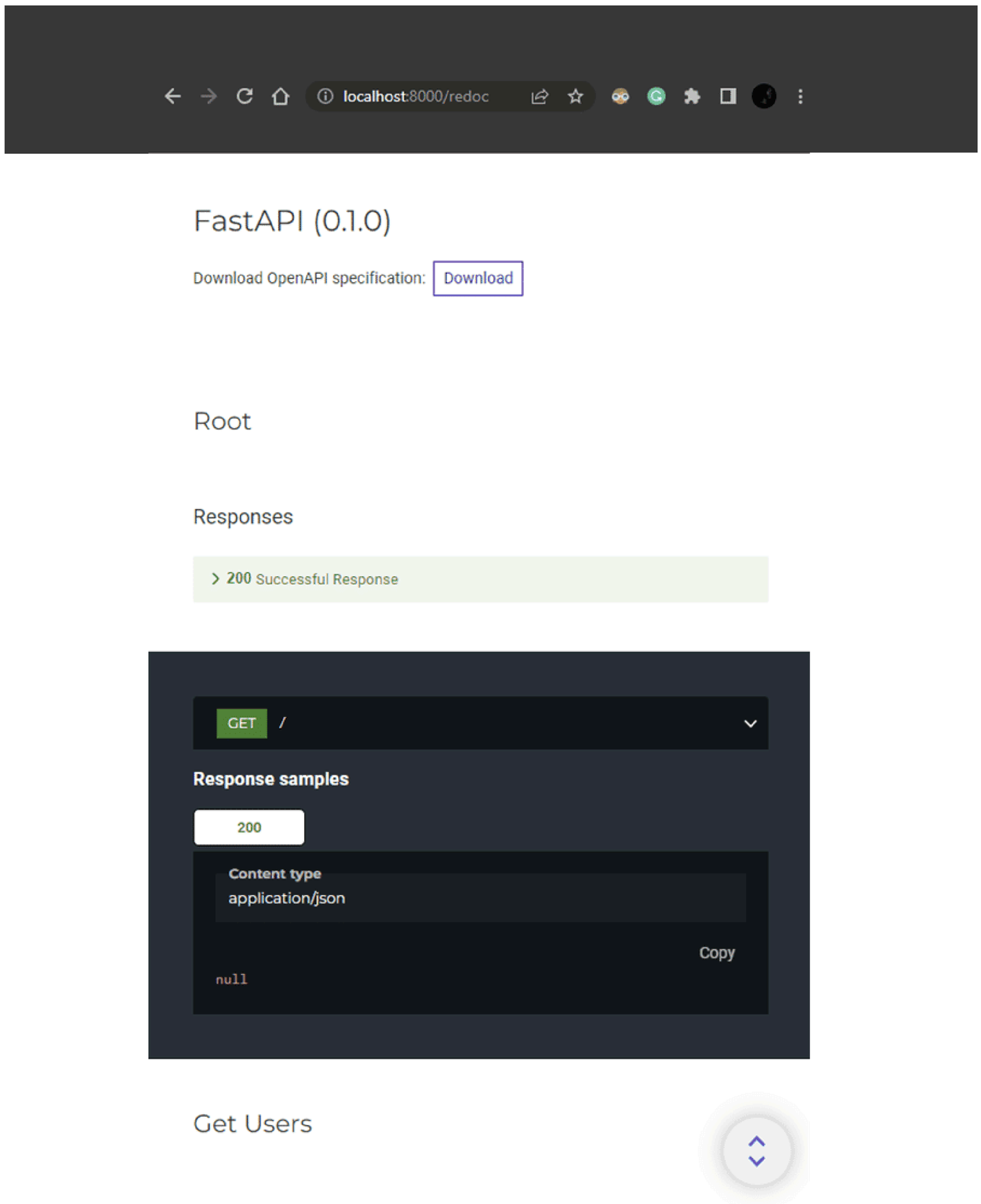
FastAPI uses Swagger UI to provide automatic interactive API documentation. To access it, navigate to `http://localhost:8000/docs` and you will see a screen with all your endpoints, methods, and schemas.




Swagger UI's documentation for FastAPI.

This automatic, browser-based API documentation is provided by FastAPI, and you don't need to do anything else to take advantage of it.

An alternative browser-based API documentation, also provided by FastAPI, is Redoc. To access Redoc, navigate to `http://localhost:8000/redoc`, where you will be presented with a list of your endpoints, the methods, and their respective responses.




Redoc's documentation for FastAPI.

**Setting Up Routes in FastAPI**

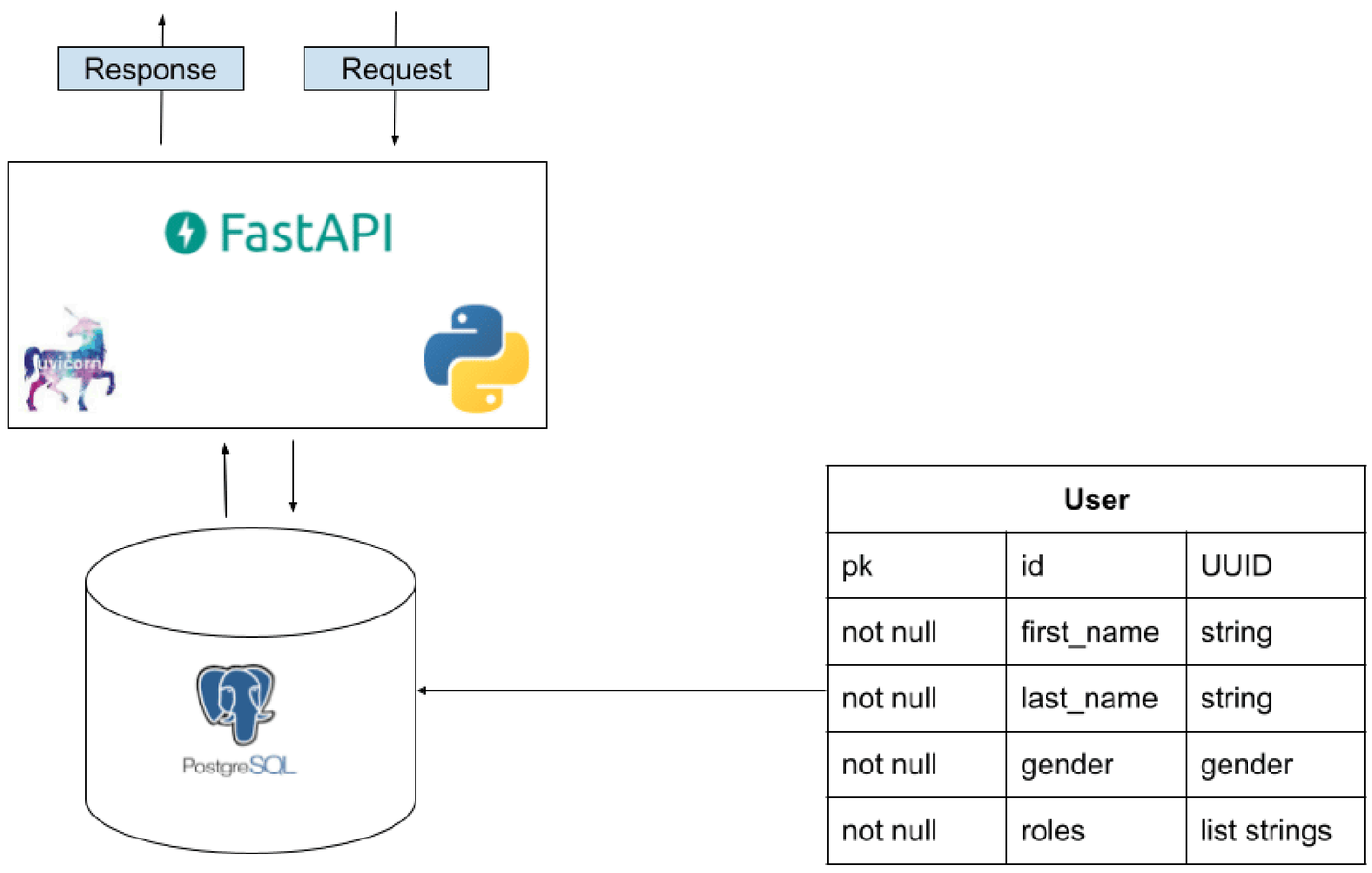
The `@app` decorator allows you to specify the route's method, such as `@app.get` or `@app.post`, and supports `GET`, `POST`, `PUT`, and `DELETE`, as well as the less common options, `HEAD`, `PATCH`, and `TRACE`.

## Building Your App With FastAPI

In this tutorial, you'll be walked through building a CRUD application with FastAPI. The application will be able to:

- Create a user
- Read a user's database record
- Update an existing user
- Delete a particular user

To execute these CRUD operations, you will create methods that expose the API endpoints. The result will be an in-memory database that can store a list of users.




Database table structure for CRUD examples.

You'll use the pydantic library to perform data validation and settings management using Python type annotations. For the purposes of this tutorial, you'll declare the shape of your data as classes with attributes.

This tutorial will use the in-memory database. This is to quickly get you started with using FastAPI to build your APIs. However, for production, you can make use of any database of your choosing, such as PostgreSQL, MySQL, SQLite, or even Oracle.

**Building the App**

You'll begin by creating your user model. The user model will have the following attributes:

- `id`: A Universal Unique Identifier (UUID)
- `first_name`: The first name of the user
- `last_name`: The last name of the user
- `gender`: The gender of the user
- `roles`, which is a list containing `admin` and `user` roles

Start by creating a new file named **models.py** in your working directory, then paste the following code into **models.py** to create your model:

```
# models.py
from typing import List, Optional
from uuid import UUID, uuid4
from pydantic import BaseModel
from enum import Enum
from pydantic import BaseModel
class Gender(str, Enum):
 male = "male"
 female = "female"
class Role(str, Enum):
 admin = "admin"
 user = "user"
class User(BaseModel):
 id: Optional[UUID] = uuid4()
 first_name: str
 last_name: str
 gender: Gender
 roles: List[Role]
```

In the code above:

- Your `User` class extends `BaseModel`, which is then imported from `pydantic`.
- You defined the attributes of the user, as discussed above.

The next step is to create your database. Replace the contents of your **main.py** file with the following code:

```
# main.py
from typing import List
from uuid import uuid4
from fastapi import FastAPI
from models import Gender, Role, User
app = FastAPI()
db: List[User] = [
 User(
 id=uuid4(),
 first_name="John",
 last_name="Doe",
 gender=Gender.male,
```

```
roles=[Role.user],
),
User(
id=uuid4(),
first_name="Jane",
last_name="Doe",
gender=Gender.female,
roles=[Role.user],
),
User(
id=uuid4(),
first_name="James",
last_name="Gabriel",
gender=Gender.male,
roles=[Role.user],
),
User(
id=uuid4(),
first_name="Eunit",
last_name="Eunit",
gender=Gender.male,
roles=[Role.admin, Role.user],
),
]
```

In **main.py**:

- You initialized db with a type of `List`, and passed in the `User` model
- You created an in-memory database with four users, each with the required attributes such as `first_name`, `last_name`, `gender`, and `roles`. The user `Eunit` is assigned the roles of `admin` and `user`, while the other three users are assigned only the role of `user`.

### Read Database Records

You have successfully set up your in-memory database and populated it with users, so the next step is to set up an endpoint that will return a list of all users. This is where FastAPI comes in.

In your **main.py** file, paste the following code just below your `Hello World` endpoint:

```
# main.py
 @app.get("/api/v1/users")
 async def get_users():
 return db
```

This code defines the endpoint `/api/v1/users`, and creates an async function, `get_users`, which returns all the contents of the database, db.

Save your file, and you can test your user endpoint. Run the following command in your terminal to start the API server:

```
$ uvicorn main:app --reload
```

In your browser, navigate to `http://localhost:8000/api/v1/users`. This should return a list of all your users, as seen below:


User data retrieved by FastAPI database read request.

At this stage, your **main.py** file will look like this:

```
# main.py
from typing import List
from uuid import uuid4
from fastapi import FastAPI
from models import Gender, Role, User
app = FastAPI()
db: List[User] = [
 User(
 id=uuid4(),
 first_name="John",
 last_name="Doe",
 gender=Gender.male,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="Jane",
 last_name="Doe",
 gender=Gender.female,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="James",
 last_name="Gabriel",
 gender=Gender.male,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="Eunit",
 last_name="Eunit",
 gender=Gender.male,
 roles=[Role.admin, Role.user],
 ),
]
@app.get("/")
async def root():
 return {"Hello": "World",}
```

```
@app.get("/api/v1/users")
async def get_users():
 return db
```

### Create Database Records

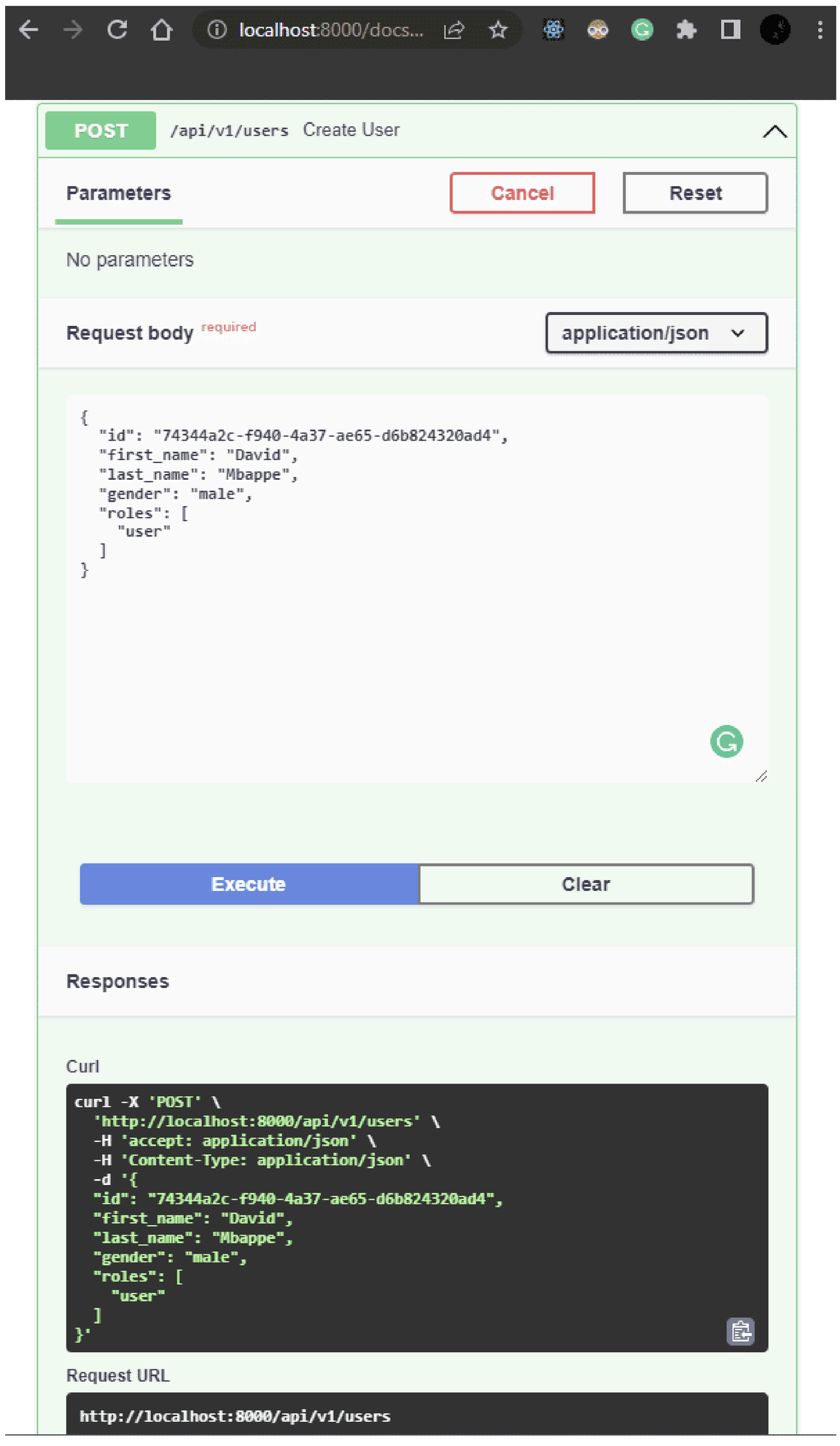
The next step is to create an endpoint to create a new user in your database. Paste the following snippet into your **main.py** file:

```
# main.py
@app.post("/api/v1/users")
async def create_user(user: User):
 db.append(user)
 return {"id": user.id}
```

In this snippet, you defined the endpoint to submit a new user and made use of the `@app.post` decorator to create a `POST` method.

You also created the function `create_user`, which accepts `user` of the `User` model, and appended (added) the newly created `user` to the database, `db`. Finally, the endpoint returns a JSON object of the newly created user's `id`.

You will have to use the automatic API documentation provided by FastAPI to test your endpoint, as seen above. This is because you cannot make a post request using the web browser. Navigate to `http://localhost:8000/docs` to test using the documentation provided by SwaggerUI.




Parameters for a FastAPI POST request.

### Delete Database Records

Since you're building a CRUD app, your application will need to have the ability to delete a specified resource. For this tutorial, you will create an endpoint to delete a user.

Paste the following code into your **main.py** file:

```
# main.py
```

```
from uuid import UUID
from fastapi HTTPException
@app.delete("/api/v1/users/{id}")
async def delete_user(id: UUID):
for user in db:
 if user.id == id:
 db.remove(user)
 return
raise HTTPException(
 status_code=404, detail=f"Delete user failed, id {id} not found."
 )
```

Here's a line-by-line breakdown of how that code works:

- `@app.delete("/api/v1/users/{id}")`: You created the delete endpoint using the `@app.delete()` decorator. The path is still `/api/v1/users/{id}`, but then it retrieves the `id`, which is a path variable corresponding to the id of the user.
- `async def delete_user(id: UUID)`:: Creates the `delete_user` function, which retrieves the `id` from the URL.
- `for user in db`:: This tells the app to loop through the users in the database, and check if the `id` passed matches with a user in the database.
- `db.remove(user)`: If the `id` matches a user, the user will be deleted; otherwise, an `HTTPException` with a status code of `404` will be raised.


Parameters for a FastAPI DELETE request.

### Update Database Records

You are going to create an endpoint to update a user's details. The details that can be updated include the following parameters: `first_name`, `last_name`, and `roles`.

In your **models.py** file, paste the following code under your `User` model, that is after the `User(BaseModel):` class:

```
# models.py
class UpdateUser(BaseModel):
first_name: Optional[str]
last_name: Optional[str]
roles: Optional[List[Role]]
```

In this snippet, the class `UpdateUser` extends `BaseModel`. You then set the updatable user parameters, such as `first_name`, `last_name`, and `roles`, to be optional.

Now you'll create an endpoint to update a particular user's details. In your **main.py** file, paste the following code after `@app.delete` decorator:

```
# main.py
@app.put("/api/v1/users/{id}")
async def update_user(user_update: UpdateUser, id: UUID):
 for user in db:
 if user.id == id:
 if user_update.first_name is not None:
 user.first_name = user_update.first_name
 if user_update.last_name is not None:
 user.last_name = user_update.last_name
 if user_update.roles is not None:
 user.roles = user_update.roles
 return user.id
 raise HTTPException(status_code=404, detail=f"Could not find user with id: {id}")
```

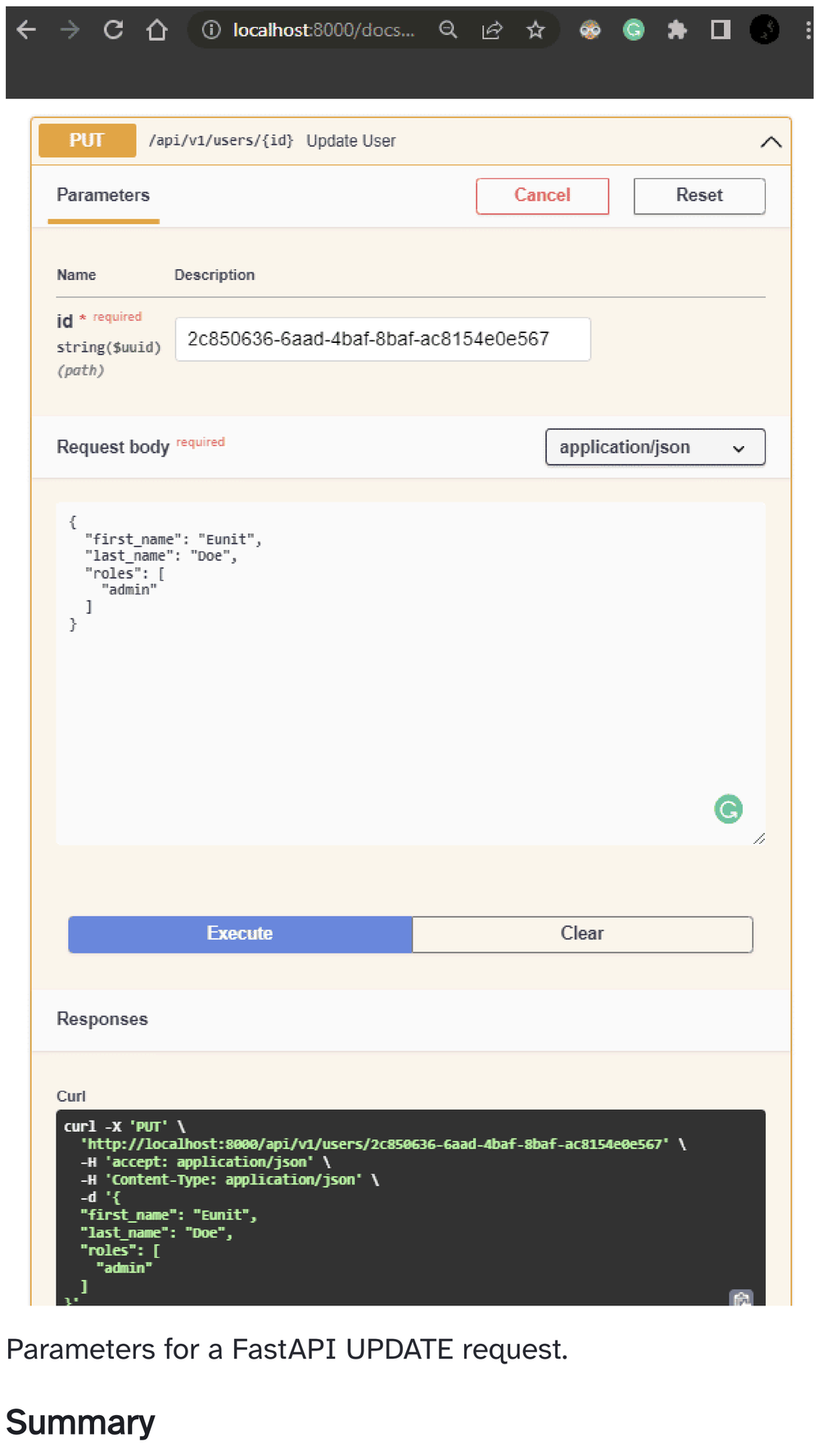
In the code above, you've done the following:

- Created `@app.put("/api/v1/users/{id}")`, the update endpoint. It has a variable parameter `id` that corresponds to the id of the user.
- Created a method called `update_user`, which takes in the `UpdateUser` class and `id`.
- Used a `for` loop to check if the user associated with the passed `id` is in the database.
- Checked if any of the user's parameters are `is not None` (not null). If any parameter, such as `first_name`, `last_name`, or `roles`, is not null, then it is updated.
- If the operation is successful, the user id is returned.
- If the user wasn't located, an `HTTPException` exception with a status code of `404` and a message of `Could not find user with id: {id}` is raised.

To test this endpoint, make sure your Uvicorn server is running. If it is not running, enter this command:

```
uvicorn main:app --reload
```

Below is a screenshot of the test.




Parameters for a FastAPI UPDATE request.

## Summary

In this tutorial, you've learned about the FastAPI framework for Python and saw for yourself how quickly you can get a FastAPI-powered application up and running. You learned how to build CRUD API endpoints using the framework — creating, reading, updating, and deleting database records.

Now, if you want to take your web app development to the next level, be sure to check out Kinsta's platform for Application Hosting and Database Hosting. Like FastAPI, it's powerfully simple.

Emmanuel Uchenna

Emmanuel is an experienced, passionate, and enthusiastic software developer and technical writer with proven years of professional experience. He focuses on full-stack web development. He's fluent in ReactJS, JavaScript, VueJS, and NodeJS and familiar with industry-standard technologies such as Git, GitHub, and TDD. He helps individuals, businesses, and brands build responsive, accessible, beautiful, and intuitive websites to improve their online presence. He is also a technical writer for various websites and his own projects.