

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Building CRUD API with FastAPI: A Step-by-Step Guide



Pradosh K · [Follow](#)

Published in DataUniverse

5 min read · Jul 29, 2023



Listen



Share

... More



*High performance, easy to learn,
fast to code, ready for production*

FastAPI is a modern and high-performance web framework for building APIs with Python. Its ease of use, speed, and support for type hints make it a popular choice for developing web services. In this tutorial, we'll create a CRUD API (Create, Read, Update, Delete) using FastAPI based on the provided code.

Introduction

In this tutorial, we'll create a simple CRUD API for managing blog posts. The API will allow us to perform the following operations:

1. Retrieve a list of all blog posts
2. Create a new blog post
3. Retrieve details of a specific blog post
4. Update an existing blog post
5. Delete a blog post

Prerequisites

- Basic knowledge of Python
- Familiarity with RESTful API concepts

Setting Up the Environment

Before we begin, make sure you have installed Python and FastAPI. You can install FastAPI and Uvicorn (ASGI server) using pip:

```
pip install fastapi uvicorn
```

Creating the CRUD API

1. Import the required modules and classes:

```
from fastapi import FastAPI, HTTPException, status, Query, Response
from pydantic import BaseModel
from typing import Optional
from random import randrange
```

2. Initialize FastAPI and Create a schema for your post

```
app = FastAPI()

class Post(BaseModel):
```

[Open in app](#)**Medium** Search**S**

Here, as you can see , we have defined a Pydantic model called `Post` , which acts as a data model for the blog post entity. Pydantic allows us to define data models with type hints and default values. The `BaseModel` class from Pydantic is used as a base class to inherit the model's functionality

Using Pydantic models to define the schema for our API data brings several benefits, such as data validation, automatic serialization/deserialization, default values, and improved code readability. It ensures that the API receives and returns data in the expected format, making the API more robust and less prone to errors.

By leveraging Pydantic models, we can handle complex data structures with ease while maintaining the simplicity and efficiency of the FastAPI framework. This combination of FastAPI and Pydantic allows us to build powerful, type-safe, and well-structured APIs in Python.

3. Create a python List containing blog post

```
my_list = [  
    {"title": "title of post 1", "content": "content of post 1", "id": 1},  
    {"title": "title of post 2", "content": "content of post 2", "id": 2}  
]
```

This above Python list containing two sample blog post dictionaries. Each dictionary represents a blog post and contains the attributes defined in the `Post` model. These sample posts are just for demonstration purposes; in a real application, this data would typically be stored in a database.

4. Implement the “Get All Posts” endpoint:

```
@app.get("/posts")  
def get_all_posts():
```

```
return {"data": my_list}
```

5. Implement the “Create Post” endpoint:

```
@app.post("/posts", status_code=status.HTTP_201_CREATED)
def create_post(post: Post):
    post_dict = post.dict()
    post_dict["id"] = randrange(0, 1000000)
    my_list.append(post_dict)
    return {"data": post_dict}
```

6. Implement the “Get Latest Post” endpoint:

```
@app.get("/posts/latest")
def get_latest_post():
    post = my_list[-1]
    return {"post_detail": post}
```

7. Implement the “Get Post by ID” endpoint:

```
@app.get("/posts/{id}")
def get_post_by_id(id: int):
    post = find_post(id)
    if not post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                             detail=f"Post with ID {id} not found")
    return {"post_detail": post}
```

8. Implement the “Delete Post” endpoint:

```
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(id: int):
    indx = find_index_post(id)
    if indx is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
```

```
detail=f"Post with ID {id} does not exist")  
my_list.pop(indx)  
return {"message": f"Post with ID {id} successfully deleted"}
```

9. Implement the “Update Post” endpoint:

```
@app.put("/posts/{id}")  
def update_post(id: int, post: Post):  
    indx = find_index_post(id)  
    if indx is None:  
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,  
                             detail=f"Post with ID {id} does not exist")  
  
    post_dict = post.dict()  
    post_dict['id'] = id  
    my_list[indx] = post_dict  
    return {"message": f"Post with ID {id} successfully updated"}
```

10. Helper functions to find post and index in the list

```
def find_post(id):  
    for p in my_list:  
        if p["id"] == id:  
            return p  
  
def find_index_post(id):  
    for i, p in enumerate(my_list):  
        if p['id'] == id:  
            return i
```

Running the API

Save the code in a file named `main.py` and run the API using Uvicorn:

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

Uvicorn is an ASGI (Asynchronous Server Gateway Interface) server that allows you to run ASGI applications, such as FastAPI, and serve them over the internet. To run the FastAPI application, you use the Uvicorn command with the following format

Now, the CRUD API will be available at `http://127.0.0.1:8000`.

Testing the Endpoints

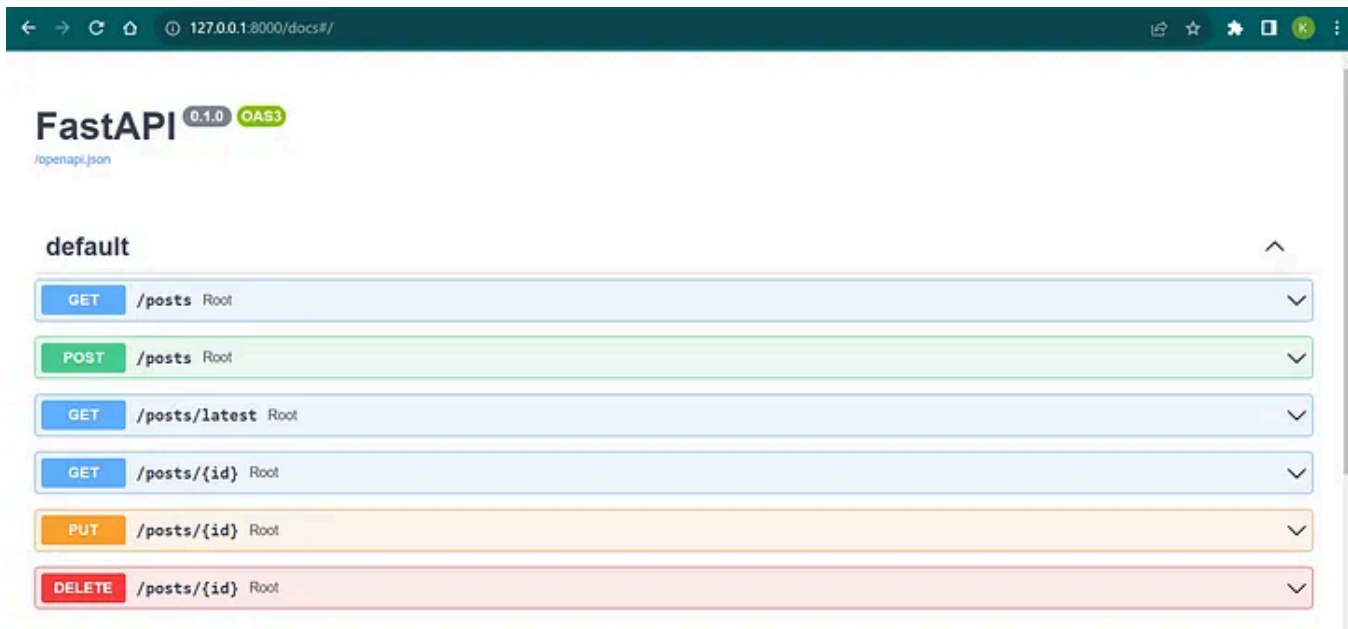
You can use Postman to do this.

1. Retrieve All Posts: Open your web browser or API client and navigate to `http://127.0.0.1:8000/posts` to retrieve all posts.
2. Create a Post: Use a POST request with JSON payload to `http://127.0.0.1:8000/posts` to create a new post.
3. Retrieve Latest Post: Visit `http://127.0.0.1:8000/posts/latest` to get the latest post.
4. Retrieve Post by ID: Access `http://127.0.0.1:8000/posts/{id}` to retrieve a specific post by ID.
5. Update Post: Use a PUT request with JSON payload to `http://127.0.0.1:8000/posts/{id}` to update an existing post.
6. Delete Post: Use a DELETE request to `http://127.0.0.1:8000/posts/{id}` to delete a post by ID.

Accessing the Interactive API Documentation

After starting the FastAPI application, you can access the interactive API documentation by visiting the `/docs` endpoint. For example, if your FastAPI server is running on `http://127.0.0.1:8000`, you can access the documentation at <http://127.0.0.1:8000/docs>.

By default, FastAPI also provides an alternative documentation page powered by ReDoc, which can be accessed at `/redoc`. For example, <http://127.0.0.1:8000/redoc>.



The automatic interactive API documentation feature of FastAPI, powered by Swagger UI and ReDoc, is a valuable tool for both developers and API consumers. It simplifies API exploration, testing, and integration by providing a user-friendly interface with detailed endpoint information and real-time request/response examples. This feature makes FastAPI an excellent choice for building robust and developer-friendly APIs.

Conclusion

In this blog, we built a simple CRUD API for managing blog posts using FastAPI. FastAPI's intuitive syntax and automatic validation make it an excellent choice for building APIs with Python. You can extend this example to implement more advanced features and connect the API to a database for persistent data storage.

FastAPI provides many other features like authentication, request validation, WebSocket support, and more. By following this tutorial, you have learned the basics of creating a CRUD API using FastAPI, and you can now explore more advanced topics and build powerful web services. Happy coding!

[Api Development](#)[API](#)[Python](#)[Fastapi](#)

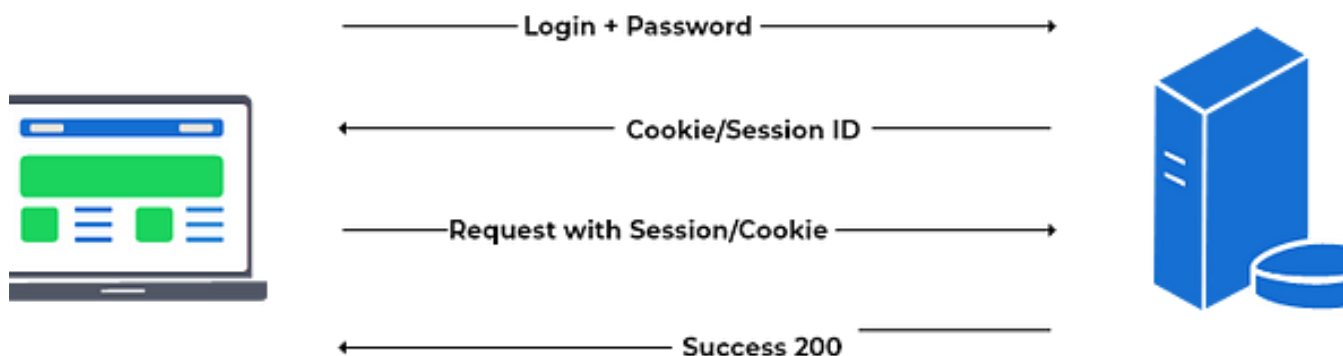
[Follow](#)

Written by Pradosh K

59 Followers · Editor for DataUniverse

More from Pradosh K and DataUniverse

Cookie/Session Based Authentication



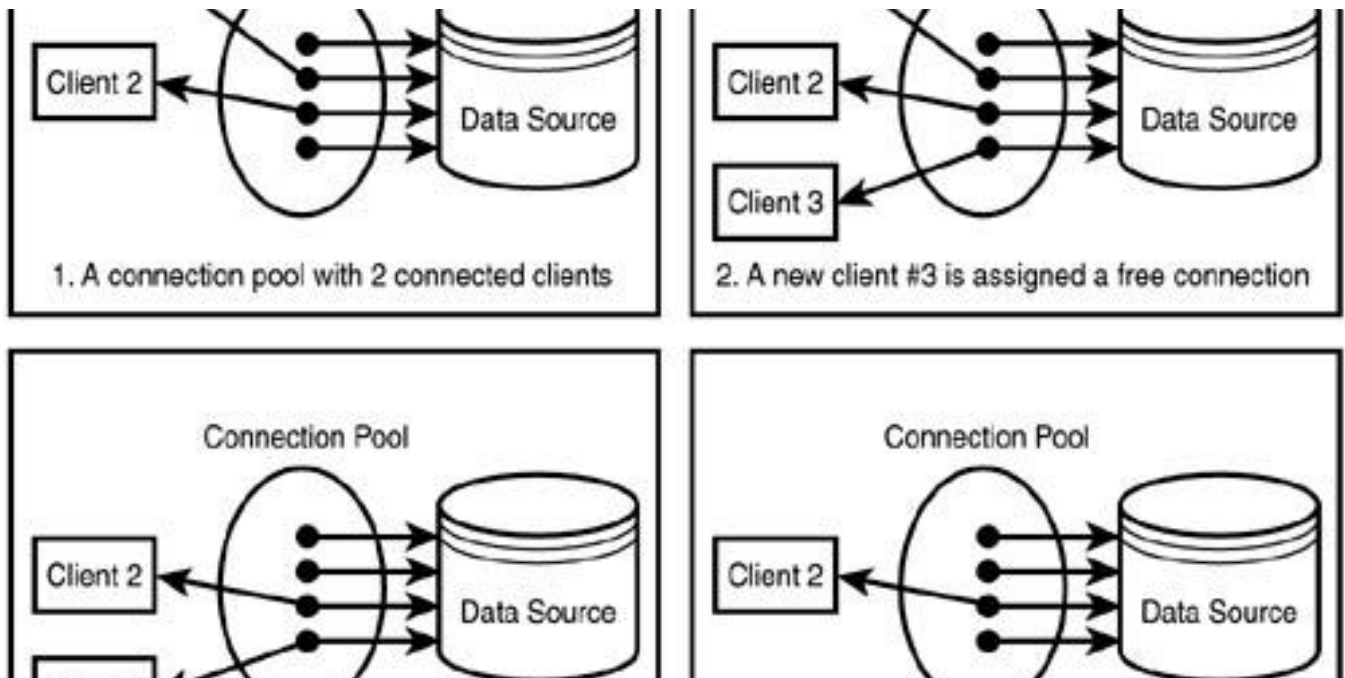
Pradosh K

Session-Based Authentication with FastAPI: A Step-by-Step Guide

If you need a quick refresher, feel free to revisit previous blog posts

Aug 5, 2023 🖱 76








 Pradosh K in DataUniverse

Optimizing Database Interaction in Web Applications: Connection Pooling with psycopg2 and...

First of all, thank you all for the overwhelming response and positive feedback on the previous blog post Building CRUD API with FastAPI: A...

Aug 3, 2023  16

   127.0.0.1:8000

```
{: "Hello World from FastAPI"}
```

 Pradosh K in DataUniverse

Introduction to FastAPI

What is an API?

Jun 3, 2023 8 1



Pradosh K in DataUniverse

How to Use Environment Variables in Python for Secure Configuration

In today's development landscape, managing configuration and sensitive information like API keys securely is paramount. Environment...

Jun 1

[See all from Pradosh K](#)[See all from DataUniverse](#)

Recommended from Medium



 Rajan Sahu


Securing Your FastAPI Application with Role-Based Authentication

In today's rapidly evolving digital landscape, security is paramount, especially when it comes to handling user data and controlling...

★ Apr 22 🖱️ 393 💬 2

🔖 ⋮



 Engr Muhammad Tanveer sultan

Building a FastAPI Application with CRUD Operations Across Multiple Databases Using SQLAlchemy.

FastAPI has quickly become one of the most popular frameworks for building APIs in Python due to its high performance and ease of use. When...

★ Aug 19 🖱 3

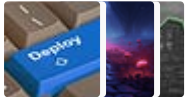


Lists



Coding & Development

11 stories · 836 saves



Predictive Modeling w/ Python

20 stories · 1582 saves



Practical Guides to Machine Learning

10 stories · 1922 saves



ChatGPT

21 stories · 827 saves



 Om Kamath in Level Up Coding

Building Vector Databases with FastAPI and ChromaDB

Beginners guide to ChromaDB vectorstores and FastAPI with Langchain

★ May 7 🖱 570 💬 5





 Amir Lavasani

How to Structure Your FastAPI Projects

Part 1: Blueprint

May 14  586  9



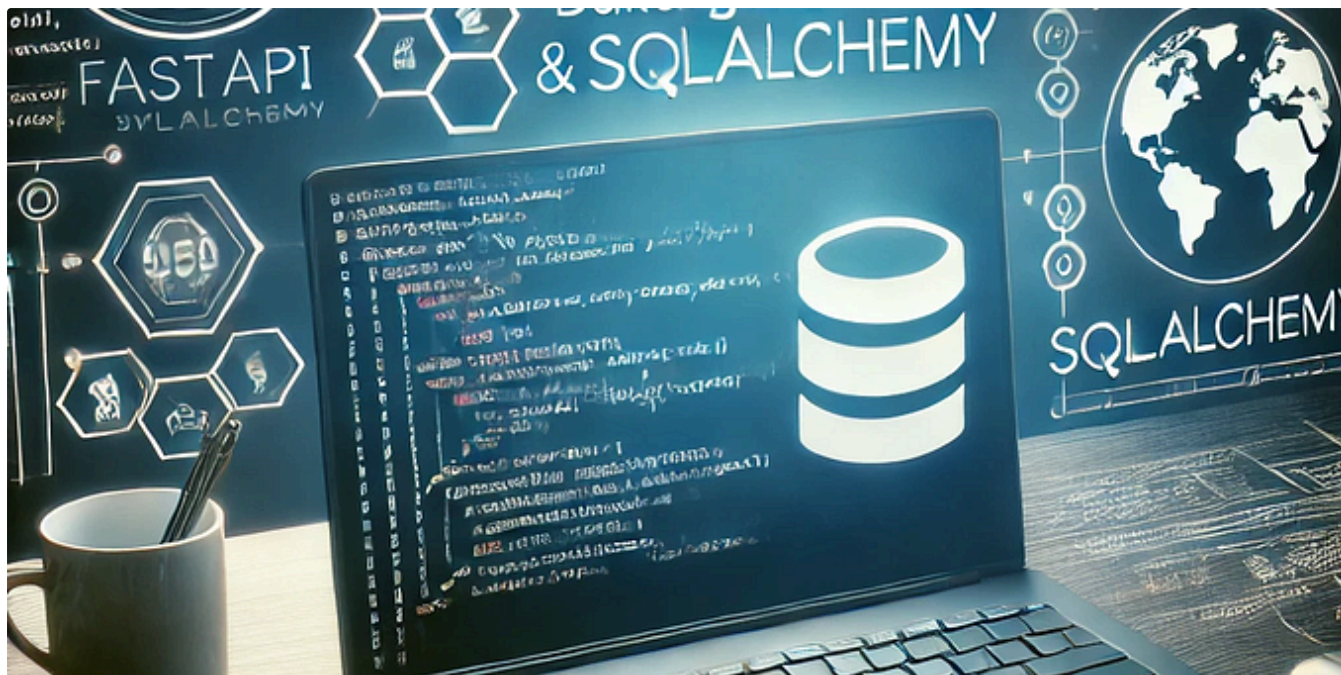
 Coding Pit

Building a Dynamic Table App with FastHTML and HTMX

A Step-by-Step Guide to Creating a CRUD Web Application in a Single View.

★ Sep 10  124





S Suganthi

FastAPI with SQLAlchemy: Building Scalable APIs with a Database Backend

APIs are the backbone of modern software applications. The faster they are, the more responsive your app becomes. With Python's ecosystem...

Sep 23 🖱 55



See more recommendations