

How to Create a RESTful API with Python and FastAPI

Learn how to build a robust RESTful API using Python and FastAPI framework with step-by-step instructions, code examples, and best practices for beginners.

Contents

| | | |
|-----|---|----|
| 01 | Introduction | 3 |
| 02 | Prerequisites | 4 |
| 03 | Set Up the FastAPI Environment | 4 |
| 04 | Create an Endpoint to List All Students | 5 |
| 05 | Add Filter Function in List Endpoint | 7 |
| 06 | Create a Function to Check if Student Object Exists | 8 |
| 07 | Create an Endpoint to Fetch Single Student Object | 9 |
| 08 | Create a Model to Handle POST/UPDATE Request Body | 10 |
| 09 | Create an Endpoint to Add New Student Object | 11 |
| 010 | Create an Endpoint to Update a Student Object | 12 |
| 011 | Create an Endpoint to Delete a Student Object | 13 |
| 012 | Final Code | 14 |
| 013 | Conclusion | 16 |

Introduction

Creating a RESTful API is a fundamental skill for modern developers, enabling seamless communication between systems, and with Python's FastAPI, you can build high-performance, reliable APIs efficiently and with minimal code.

FastAPI is a Python web framework for building APIs & web applications. It supports modern Python features like asynchronous processing & type hints, making it fast and efficient. In addition, it uses the Asynchronous Server Gateway Interface (ASGI) standard for asynchronous, concurrent client connectivity, and it can work with WSGI if needed.

API is an acronym for Application Programming Interface. It acts as a mediator, allowing applications to talk to each other, becoming a medium between applications to exchange data or trigger action. While an API can communicate over many different protocols, this article refers to Web APIs that communicate over HTTP protocol.

REST (Representational State Transfer) is a set of architectural constraints used to build an API, not a protocol or a standard. It does not dictate exactly how to create an API. Instead, it introduces best practices known as constraints. A **RESTful API** is an API that complies with the REST architecture.

Any request made to a REST API consists of four essential parts: method, endpoint, headers, and body.

1. **HTTP method** describes action or operation.
 - **POST** - Create a resource
 - **GET** - Retrieve a resource
 - **PUT** - Update a resource
 - **DELETE** - Delete a resource
2. **Endpoint** contains a URI (uniform resource identifier) used to locate the resource on the web.
3. **Headers** contain authentication information such as the API key.

4. **Body** contains the data or any additional information.

This article demonstrates the steps to build a basic RESTful API using Python and FastAPI. The presented application allows users to access and perform CRUD operations on objects stored in the database. Following is the table for reference showing the endpoint structure of the given application.

| Endpoint | Method | Description |
|------------------------|---------------|--------------------------|
| /students | GET | List all student objects |
| /students | POST | Add new student object |
| /students/{student_id} | GET | Return a student object |
| /students/{student_id} | PUT | Update a student object |
| /students/{student_id} | DELETE | Delete a student object |

Prerequisites

- Deploy a fresh [Ubuntu](#) Server
- Create non-root sudo user

Set Up the FastAPI Environment

1. Initialize the project directory.

Create a folder for your project.

```
$ mkdir ~/restful_demo
```

Open the project directory.

```
$ cd ~/restful_demo
```

2. Create a virtual environment.

A virtual environment is a Python tool for dependency management and project isolation. Each project can have any package installed locally in an isolated directory.

Install the Python virtual environment package.

```
$ sudo apt update  
$ sudo apt install python3-venv
```

Create a Python virtual environment for your project.

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

Enter the virtual environment.

```
$ source venv/bin/activate
```

Exit the virtual environment.

```
(venv) $ deactivate
```

3. Install the dependencies.

Install the `wheel` Python package.

```
(venv) $ pip install wheel
```

Install the `fastapi` Python package.

```
(venv) $ pip install fastapi[all]
```

Create an Endpoint to List All Students

Create a new file named `app.py`.

```
$ nano app.py
```

Import the `FastAPI` class from the `fastapi` library and assign an instance to a variable named `app`.

```
from fastapi import FastAPI

app = FastAPI()
```

Typically, a REST API uses a database server to store data, such as PostgreSQL or MongoDB. As database choice is dependent on user preference, this article has demonstrated the steps using a Python list as an in-memory database. You can implement it with any database using the same principles.

Create an in-memory database by assigning a list of dummy student objects to a variable named `students`.

```
students = [
    {'name': 'Student 1', 'age': 20},
    {'name': 'Student 2', 'age': 18},
    {'name': 'Student 3', 'age': 16}
]
```

Create a route to handle **GET** requests on endpoint `/students`.

```
@app.get('/students')
def user_list():
    return {'students': students}
```

`@app.get('/students')` decorator tells FastAPI to execute `user_list` function, which returns all the student objects stored in the database. This route enables users to list all the available student objects by sending a **GET** request on the `/students` endpoint.

Save the file using Ctrl + X then Enter.

Deploy the FastAPI application using `uvicorn`.

```
(venv) $ uvicorn app:app --debug
```

Send a test request in a new terminal.

```
curl http://127.0.0.1:8000/students
```

Expected Output:

```
{"students": [{"name": "Student 1", "age": 20}, {"name": "Student 2", "age": 18}, {"name": "Student 3", "age": 16}]}
```

Stop the `uvicorn` server using `Ctrl + C`.

Add Filter Function in List Endpoint

This section demonstrates the steps to update the list endpoint and add functionality to filter the output based on the range of age provided by the user.

Open the FastAPI application created in the previous section.

```
$ nano app.py
```

Import `Optional` class from `typing` library.

```
from typing import Optional
```

Place the given line below the first line where you imported the `FastAPI` class.

Update the `user_list` function to accept min/max parameters and rearrange the code inside the function to return the filtered list.

```
@app.get('/students')
def user_list(min: Optional[int] = None, max: Optional[int] = None):

    if min and max:

        filtered_students = list(
            filter(lambda student: max >= student['age'] >= min, students)
        )
```

```
        return {'students': filtered_students}

    return {'students': students}
```

The modifications made to the list endpoint enable users to pass a range of ages using min and max parameters to filter the list of student objects. The `user_list` function takes two optional parameters: min and max. The in-built Python function `filter` iterates over each student object in the database, checking if they are between the requested range. The endpoint returns all student objects stored in the database if min/max parameters are missing.

Save the file using Ctrl + X then Enter.

Deploy the FastAPI application using `uvicorn`.

```
(venv) $ uvicorn app:app --debug
```

Send a test request in a new terminal.

```
curl "http://127.0.0.1:8000/students?min=16&max=18"
```

Expected Output:

```
{"students": [{"name": "Student 2", "age": 18}, {"name": "Student 3", "age": 16}]}
```

Stop the `uvicorn` server using Ctrl + C.

Create a Function to Check if Student Object Exists

This section demonstrates the steps to create a function named `user_check`, which checks if the requested user exists in the database.

Open the FastAPI application.

```
$ nano app.py
```

Import the `HTTPException` class from `fastapi` library.

```
from fastapi import FastAPI, HTTPException
```

Edit the line where you imported `FastAPI` class to import another class.

Create a new function named `student_check` which takes `student_id` parameter.

```
def student_check(student_id):  
    if not students[student_id]:  
        raise HTTPException(status_code=404, detail='Student Not Found')
```

In the following steps, routes to perform operations on a single student object use the `student_check` function to validate if the `student_id` provided exists in the database.

Create an Endpoint to Fetch Single Student Object

This section demonstrates the steps to add a new endpoint enabling users to fetch a single student object from the database.

Open the FastAPI application.

```
$ nano app.py
```

Create a route to handle **GET** requests on endpoint `/students/{student_id}`.

```
@app.get('/students/{student_id}')  
def user_detail(student_id: int):  
    student_check(student_id)  
    return {'student': students[student_id]}
```

`@app.get('/students/{student_id}')` decorator tells FastAPI to execute `user_detail` function, which returns all the student object requested by the user. This route enables users to fetch a single student object by sending a **GET** request on

the `/students/{student_id}` endpoint. If the `student_id` provided by the user is not present in the index of the `students` list (the in-memory database), then the function returns HTTP Exception with a 404 error code.

Save the file using `Ctrl + X` then `Enter`.

Deploy the FastAPI application using `uvicorn`.

```
(venv) $ uvicorn app:app --debug
```

Send a test request in a new terminal.

```
curl http://127.0.0.1:8000/students/0
```

Expected Output:

```
{"student":{"name":"Student 1","age":20}}
```

Stop the `uvicorn` server using `Ctrl + C`.

Create a Model to Handle POST/UPDATE Request Body

This section demonstrates the steps to add a new class to define the schema for handling the request body for POST/UPDATE requests.

Open the FastAPI application created in the previous section.

```
$ nano app.py
```

Import `BaseModel` class from `pydantic` library.

```
from pydantic import BaseModel
```

Place the given line below the second line where you imported the `Optional` class.

Inherit a new class named `Student` from `BaseModel` and add the required properties.

```
class Student(BaseModel):  
    name: str  
    age: int
```

In the following steps, routes to add/update a student object use this schema to validate the values provided by the user.

Create an Endpoint to Add New Student Object

This section demonstrates the steps to add a new endpoint enabling users to add a new student object to the database.

Open the FastAPI application.

```
$ nano app.py
```

Create a route to handle **POST** requests on endpoint `/students/{student_id}`.

```
@app.post('/students')  
def user_add(student: Student):  
    students.append(student)  
  
    return {'student': students[-1]}
```

`@app.post('/students')` decorator tells FastAPI to execute the `user_add` function, which adds the provided student object to the database after validating it using the `Student` schema and returns the newly added student object. This route enables users to add a new student object in the database by sending a **POST** request on the `/students` endpoint with data containing the student object to add.

Save the file using `Ctrl + X` then `Enter`.

Deploy the FastAPI application using `uvicorn`.

```
(venv) $ uvicorn app:app --debug
```

Send a test request in a new terminal.

```
curl -X 'POST' http://127.0.0.1:8000/students -H 'Content-Type: application/json' -d '{"name":"New Student", "age": 24}'
```

Expected Output:

```
{"student":{"name":"New Student","age":24}}
```

Stop the `uvicorn` server using `Ctrl + C`.

Create an Endpoint to Update a Student Object

This section demonstrates the steps to add a new endpoint enabling users to update an existing student object in the database.

Open the FastAPI application.

```
$ nano app.py
```

Create a route to handle **PUT** requests on endpoint `/students/{student_id}`.

```
@app.put('/students/{student_id}')
def user_update(student: Student, student_id: int):
    student_check(student_id)
    students[student_id].update(student)

    return {'student': students[student_id]}
```

`@app.put('/students/{student_id}')` decorator tells FastAPI to execute `user_update` function, which updates the student object in the database by using provided

`student_id` as index after validating it using the `Student` schema and returns the newly updated student object. This route enables users to update an existing student object by sending a **PUT** request on the `/students/{student_id}` endpoint with data containing the student object to update. If the `student_id` provided by the user is not present in the index of the `students` list (the in-memory database), then the function returns HTTP Exception with a 404 error code.

Save the file using `Ctrl + X` then `Enter`.

Deploy the FastAPI application using `uvicorn`.

```
(venv) $ uvicorn app:app --debug
```

Send a test request in a new terminal.

```
curl -X 'PUT' http://127.0.0.1:8000/students/0 -H 'Content-Type: application/json' -d '{"name":"Student X", "age": 18}'
```

Expected Output:

```
{"student":{"name":"Student X","age":18}}
```

Stop the `uvicorn` server using `Ctrl + C`.

Create an Endpoint to Delete a Student Object

This section demonstrates the steps to add a new endpoint enabling users to delete a student object from the database.

Open the FastAPI application.

```
$ nano app.py
```

Create a route to handle **DELETE** requests on endpoint `/students/{student_id}`.

```
@app.delete('/students/{student_id}')
def user_delete(student_id: int):
    student_check(student_id)
    del students[student_id]

    return {'students': students}
```

`@app.delete('/students/{student_id}')` decorator tells FastAPI to execute `user_delete` function, which deleted the request student object in the database by using provided `student_id` as index. This route enables users to delete a student object from the database by sending a **DELETE** request on the `/students/{student_id}` endpoint. If the `student_id` provided by the user is not present in the index of the `students` list (the in-memory database), then the function returns HTTP Exception with a 404 error code.

Save the file using `Ctrl + X` then `Enter`.

Deploy the FastAPI application using `uvicorn`.

```
(venv) $ uvicorn app:app --debug
```

Send a test request in a new terminal.

```
curl -X 'DELETE' http://127.0.0.1:8000/students/0
```

Expected Output:

```
{"students":[{"name":"Student 2","age":18},{"name":"Student 3","age":16}]}
```

Stop the `uvicorn` server using `Ctrl + C`.

Final Code

Here is the final demonstration application code. You can use it as a reference for troubleshooting.

```
from fastapi import FastAPI, HTTPException
from typing import Optional
from pydantic import BaseModel

app = FastAPI()

students = [
    {'name': 'Student 1', 'age': 20},
    {'name': 'Student 2', 'age': 18},
    {'name': 'Student 3', 'age': 16}
]

class Student(BaseModel):
    name: str
    age: int

@app.get('/students')
def user_list(min: Optional[int] = None, max: Optional[int] = None):

    if min and max:

        filtered_students = list(
            filter(lambda student: max >= student['age'] >= min, students)
        )

        return {'students': filtered_students}

    return {'students': students}

@app.get('/students/{student_id}')
def user_detail(student_id: int):
    student_check(student_id)
    return {'student': students[student_id]}

@app.post('/students')
def user_add(student: Student):
    students.append(student)

    return {'student': students[-1]}

@app.put('/students/{student_id}')
def user_update(student: Student, student_id: int):
    student_check(student_id)
```

```
students[student_id].update(student)

return {'student': students[student_id]}

@app.delete('/students/{student_id}')
def user_delete(student_id: int):
    student_check(student_id)
    del students[student_id]

    return {'students': students}

def student_check(student_id):
    if not students[student_id]:
        raise HTTPException(status_code=404, detail='Student Not Found')
```

Conclusion

In this article, you learned the basics of REST API and how to create one yourself using Python and FastAPI. To deploy your FastAPI applications to production, follow the steps to [deploy FastAPI Applications with Gunicorn and Nginx on Ubuntu 20.04](#). For more information related to FastAPI, visit the official [FastAPI website](#).



VULTR

