**FAST API**

**Introduction**

In the world of modern web development, APIs play a crucial role in enabling communication between different software systems. FastAPI, a high-performance Python web framework, is rapidly gaining popularity due to its ease of use, speed, and robustness. This project demonstrates a simple yet powerful CRUD (Create, Read, Update, Delete) API built using FastAPI, showcasing the framework’s features and potential.

This documentation provides an in-depth look at the project’s structure, key components, and future enhancements, serving as a comprehensive guide for developers interested in building scalable APIs with FastAPI.

**Project Overview**

The primary goal of this project is to create a RESTful API for managing a database of student records. The API exposes several endpoints that allow clients to perform standard CRUD operations:

Create a new student record  
 Retrieve all student records  
 Retrieve a single student record by ID  
 Update an existing student record  
 Delete a student record

The project uses an in-memory database (a simple Python list) to store student record records. This approach keeps things lightweight and straightforward, making it ideal for demonstration and learning. However, in real-world applications, this would be replaced by a robust database system like PostgreSQL or MongoDB.

**Key Technologies**

The project leverages several modern technologies and tools:

* FastAPI: A modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints.
* Pydantic: Used for data validation and serialization/deserialization. It ensures data coming into the API endpoints meets the expected structure.
* Uvicorn: An ASGI server used to run the FastAPI application, supporting asynchronous request handling.
* Python 3.7+: The programming language used to build this project.
* Swagger UI and ReDoc: Auto-generated documentation interfaces provided by FastAPI for testing and exploring the API.

These technologies combine to deliver a high-performance, developer-friendly API that can scale well with demand.

**Project Structure**

For simplicity, the project uses a single file approach (main.py). However, in production, you would typically use a multi-file structure to separate concerns:

* main.py # Main application file
* models.py # Data models (Pydantic schemas)
* routes/ # Route definitions and logic
* database.py # Database connection and ORM models
* README.md # Project description

In this demonstration, everything is contained within main.py.

**Project Code:**

**Main.py**

# from fastapi import FastAPI, Depends, HTTPException  
# from pydantic import BaseModel  
# app = FastAPI()  
#  
# students = []  
#  
# class Student(BaseModel):  
# name: str  
# age: int  
# grade: int  
# @app.get("/")  
# def read\_root():  
# return {"Message": "Welcome to FastAPI!"}  
#  
# @app.get("/students")  
# def get\_students():  
# return students  
# @app.head("/students")  
# def head\_students():  
# return {"X-Total students": len(students)}  
#  
# @app.options("/students")  
# def options\_students():  
# return {  
# "allowed\_methods": ["GET", "POST", "PUT", "PATCH", "DELETE", "OPTIONS", "HEAD"]  
# }  
# @app.post("/students")  
# def create\_student(student: Student):  
# students.append(student.dict())  
# return {"message": "Student added","data" : Student}  
#  
# @app.put("/students/{student\_id}")  
# def get\_student(student\_id: int):  
# if 0 <= student\_id < len(students):  
# return students[student\_id]  
# raise HTTPException(status\_code=404, detail="Student not found")  
#  
# @app.put("/students/{student\_id}")  
# def update\_student(student\_id: int, student: Student):  
# if 0 <= student\_id < len(students):  
# students[student\_id] = student.dict()  
# return {"Message": "Student updated","data" : students}  
# raise HTTPException(status\_code=404, detail="Student not found")  
#  
#  
# @app.patch("/students/{student\_id}")  
# def partial\_update\_student(student\_id: int,student: Student):  
# if 0 <= student\_id < len(students):  
# current\_data = students[student\_id]  
# update\_data = student.dict(exclude\_unset=True)  
# current\_data.update(update\_data)  
# student[student\_id] = current\_data  
# return {"Message": "Student partially updated","data" : current\_data}  
# raise HTTPException(status\_code=404, detail="Student not found")  
#  
# @app.delete("/students/{student\_id}")  
# def delete\_student(student\_id: int):  
# if 0 <= student\_id < len(students):  
# removed = students.pop(student\_id)  
# return {"Message": "Student removed","data" : removed}  
# raise HTTPException(status\_code=404, detail="Student not found")  
#  
# @app.get("/search")  
# def search\_students(name:str = None):  
# if name:  
# results = [s for s in students if s["name"].lower() == name.lower()]  
# return { "results":results}  
# return { "message": "no name found" }  
#  
# def common\_dependency():  
# return {"note": "common dependency injected"}  
#  
# @app.get(":/check")  
# def check(dep=Depends(common\_dependency)):  
# return dep  
# from fastapi import FastAPI, HTTPException, Request, status  
# from pydantic import BaseModel  
# from typing import Optional  
# from fastapi.responses import JSONResponse  
#  
# app = FastAPI()  
#  
# students = []  
#  
#  
# # Pydantic model for full data (POST)  
# class Student(BaseModel):  
# name: str  
# age: int  
# grade: str  
#  
#  
# # Pydantic model for partial data (PATCH)  
# class UpdateStudent(BaseModel):  
# name: Optional[str] = None  
# age: Optional[int] = None  
# grade: Optional[str] = None  
#  
#  
# # Custom exception class  
# class StudentNotFound(Exception):  
# def \_\_init\_\_(self, student\_id: int):  
# self.student\_id = student\_id  
#  
#  
# # Custom exception handler  
# @app.exception\_handler(StudentNotFound)  
# def student\_not\_found\_handler(request: Request, exc: StudentNotFound):  
# return JSONResponse(  
#  
#  
# status\_code=404,  
# content={"message": f"Student with ID {exc.student\_id} not found."},  
# )  
#  
#  
# # POST - Add a new student  
# @app.post("/students", status\_code=status.HTTP\_201\_CREATED)  
# def create\_student(student: Student):  
# students.append(student.model\_dump())  
# return {  
# "message": "Student added successfully",  
# "data": student  
# }  
#  
#  
# # PATCH - Update existing student partially  
# @app.patch("/students/{student\_id}")  
# def partial\_update\_student(student\_id: int, student: UpdateStudent):  
# if student\_id < 0 or student\_id >= len(students):  
# raise StudentNotFound(student\_id)  
#  
# updated\_data = student.model\_dump(exclude\_unset=True)  
# students[student\_id].update(updated\_data)  
#  
# return {  
# "message": "Student updated successfully",  
# "data": students[student\_id]  
# }  
#  
#  
# # GET - Retrieve all students (optional utility)  
# @app.get("/students")  
# def get\_all\_students():  
# return {  
# "message": "List of all students",  
# "data": students  
# }  
#  
# from fastapi import FastAPI, HTTPException  
# from sqlmodel import Session, select  
# from models import Student  
# from database import engine, create\_db\_and\_tables  
#  
# app=FastAPI()  
#  
# @app.on\_event("startup")  
# def on\_startup():  
# create\_db\_and\_tables()  
#  
# @app.on\_event("/students")  
# def add\_student(student: Student):  
# with Session(engine) as session:  
# session.add(student)  
# session.commit()  
# session.refresh(student)  
# return student  
#  
# @app.get("/students")  
# def get\_students():  
# with Session(engine) as session:  
# statement = select(Student)  
# students = session.exe(statement).all()  
# return students  
from http.client import HTTPException  
  
from fastapi import FastAPI  
from fastapi.responses import JSONResponse  
from sqlmodel import Field, SQLModel, Session, create\_engine, select  
from cachetools import TTLcache  
import logging  
  
logging.basicConfig(level=logging.INFO)  
  
app=FastAPI()  
  
class Student(SQLModel, table=True):  
 id: int = Field(default=None, primary\_key=True)  
 name: str  
 age: int  
 grade: str  
  
sqlite\_file\_name="students.db"  
engine = create\_engine(f"sqlite:///{sqlite\_file\_name}", echo=False)  
  
def create\_db\_and\_tables():  
 SQLModel.metadata.create\_all(engine)  
  
@app.on\_event("startup")  
def on\_startup():  
 create\_db\_and\_tables()  
  
cache = TTLcache(maxsize=100, ttl=30)  
  
@app.get("/students")  
async def get\_students(version: str = Header(default="v1")):  
 logging.info("Fetching students list")  
  
 if 'students' in cache:  
 logging.info("Serving from cache")  
 return cache["students"]  
  
 with Session(engine) as session:  
 statement = select(Student)  
 result = session.exec(statement).all()  
  
 students\_with\_links=[]  
 for students in results:  
 students\_with\_links.append({  
 "id": students.id,  
 "name": students.name,  
 "age": students.age,  
 "grade": students.grade,  
 })  
 cache["students"] = students\_with\_links  
 return students\_with\_links  
  
@app.post("/students")  
def add\_student(student: Student):  
 with Session(engine) as session:  
 existing = session.exec(select(Student).where(Student.id == student.id)).first()  
 if existing:  
 raise HTTPException(status\_code=409, detail="Student with this ID already exists")  
  
@app.get("/v1/students")  
def v1\_students():  
 return {"version": "v1", "message": "Using v2 structure with new features"}  
  
@app.get("/v2/students")  
def v2\_students():  
 return {"version": "v2", "message": "Using v2 structure with new features"}  
  
@app.get("/students-deprecated")  
def deprecated\_students():  
 return JSONResponse(  
 content={"message": "This endpoint is deprecated. please use /students"},  
 headers={"Deprecation": "true"}  
 )  
  
@app.get("/students/{student\_id}")  
def get\_student(student\_id: int):  
 with Session(engine) as session:  
 student = session.get(Student, student\_id)  
 if not student:  
 raise HTTPException(status\_code= 404, details = "student not found")  
 return student  
  
@app.get("/students/{student\_id}")  
def update\_student(student\_id: int, updated\_data: Student):  
 with Session(engine) as session:  
 student = session.get(Student, student\_id)  
 if not student:  
 raise HTTPException(status\_code=404, detail="Student not found")  
 student.name = updated\_data.name  
 student.age = updated\_data.age  
 student.grade = updated\_data.grade  
 session.add(student)  
 session.commit()  
 session.refresh(student)  
 return student  
  
@app.delete("/students/{student\_id}")  
def delete\_student(student\_id: int):  
 with Session(engine) as session:  
 student = session.get(Student, student\_id)  
 if not student:  
 raise HTTPException(status\_code=404, detail="Student not found")  
 session.delete(student)  
 session.commit()  
 return {"message": f"Student {student\_id} deleted"}

**Models.py**

from sqlmodel import SQLModel, Field  
from typing import Optional  
  
class Student(SQLModel, table=True):  
 id: Optional[int] = Field(default=None, primary\_key=True)  
 name: str  
 age: int  
 grade: str

**Test.py**

from fastapi.testclient import TestClient  
from main import app  
  
client = TestClient(app)  
  
def test\_create\_student\_success():  
 response = client.post("/students",json={  
 "name":"Alice",  
 "age":22,  
 "grade":"A"  
 })  
 assert response.status\_code == 201  
 assert response.json()["message"] == "Student added successfully"  
  
def test\_get\_all\_students():  
 response = client.get("/students")  
 print("Status code:", response.status\_code)  
 print("Response JSON:", response.json())  
 assert response.status\_code == 200  
 assert "data" in response.json()  
  
def test\_partial\_update\_student\_success():  
 response=client.patch("/students/0",json={  
 "age":22  
 })  
 assert response.status\_code == 200  
 assert response.json()["message"] == "Student updated successfully"  
 assert response.json()["data"]["age"] == 22  
 print("Status code:", response.status\_code)  
 print("Response JSON:", response.json())  
  
def test\_partial\_update\_invalid\_id():  
 response=client.patch("/students/999", json={"age":30})  
 assert response.status\_code == 404  
 assert "Student with ID" in response.json()["message"]  
 print("Status code:", response.status\_code)  
  
def test\_create\_student\_invalid\_data():  
 response = client.post("/students",json={  
 "name":"Bob",  
 })  
 assert response.status\_code == 422  
 print("Status code:", response.status\_code)  
 print("Response JSON:", response.json())  
  
def test\_patch\_student\_invalid\_data():  
 response = client.patch("/students/0",json={  
 "age":"twenty"  
 })  
 assert response.status\_code == 422  
 print("Status code:", response.status\_code)  
 print("Response JSON:", response.json())

**Database.py**

from sqlmodel import SQLModel, create\_engine  
  
DATABASE\_URL = "sqlite:///students.db"  
  
engine = create\_engine(DATABASE\_URL,echo=True)  
  
def create\_db\_and\_tables():  
 SQLModel.metadata.create\_all(engine)

**INSTALLATION PROCESS:**

**Now, Install Python 3.10 and Install the efficient dependencies like: pip, fastapi, SqlModel, uvicorn**

As we first need to install python from

* choco install python

(or)

* install python from <https://www.python.org/downlods>
* Download latest version
* Install pip by

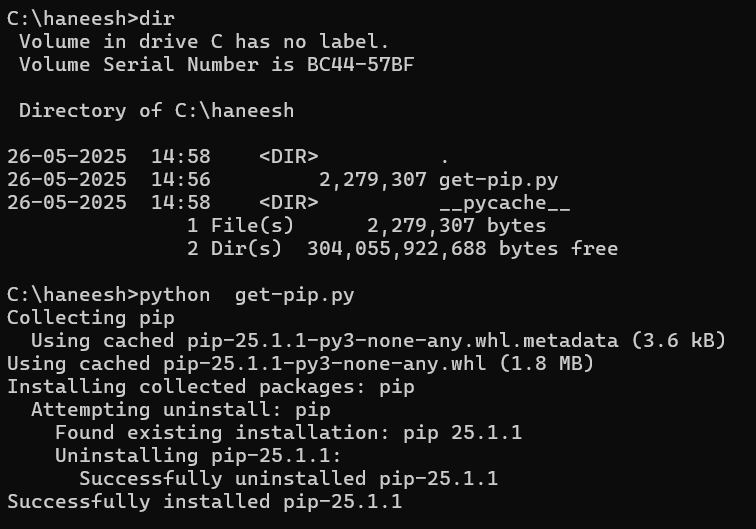
Downloading script from <https://bootstrap.pypa.io/get-pip.py>

And run it on terminal by opening the file path in terminal

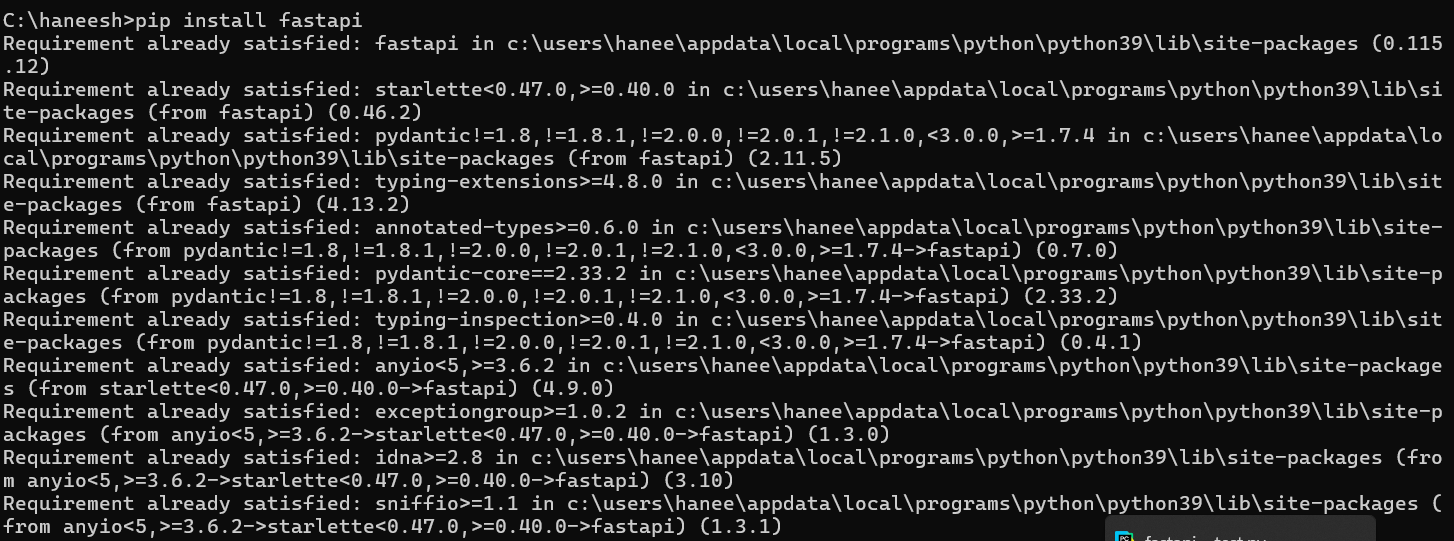
Command: python get-pip.py

**STEPS:**

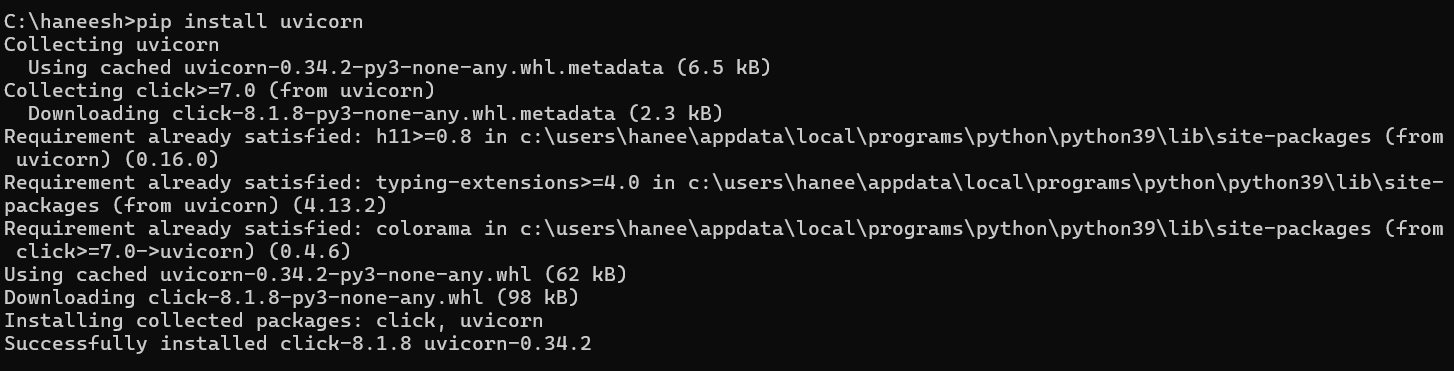
1. Install get-pip.py



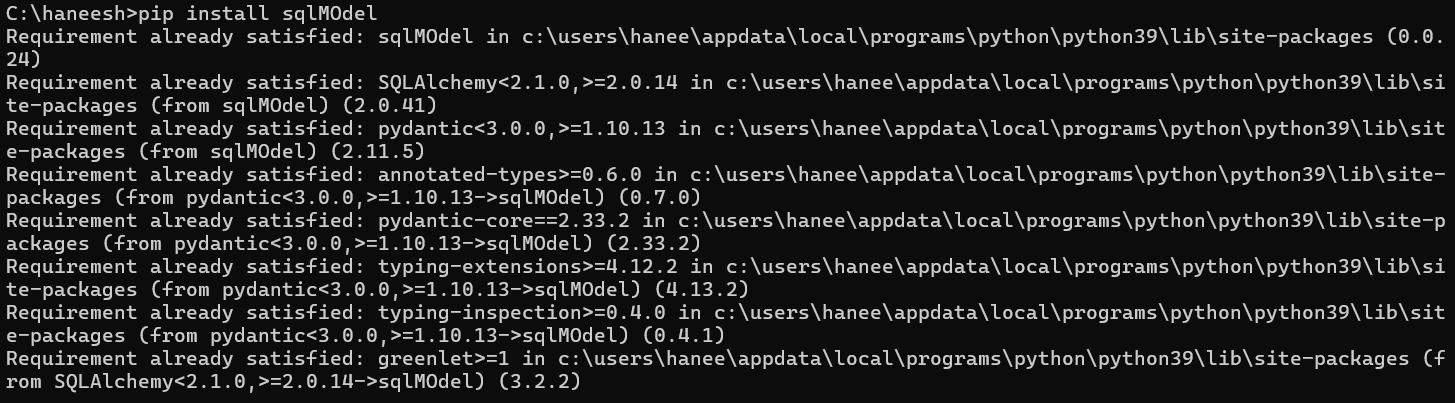
1. Install fastapi



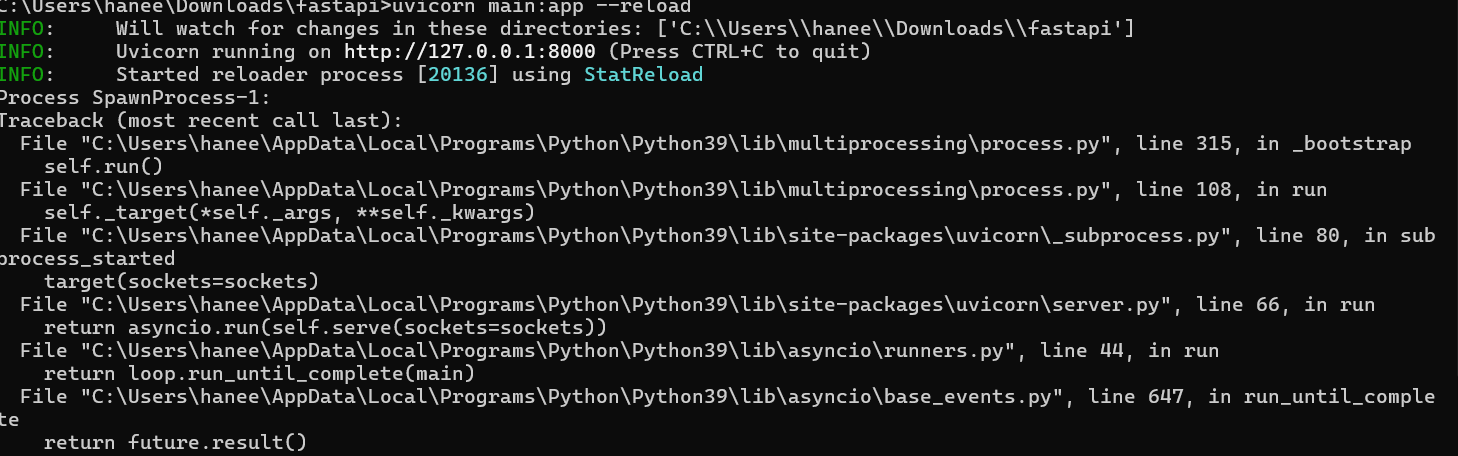
1. Install uvicorn



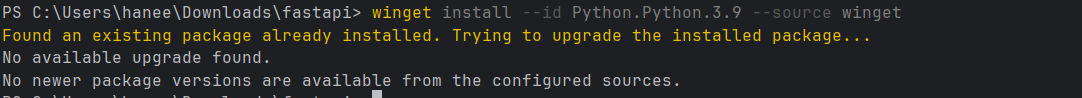
1. Install sqlModel



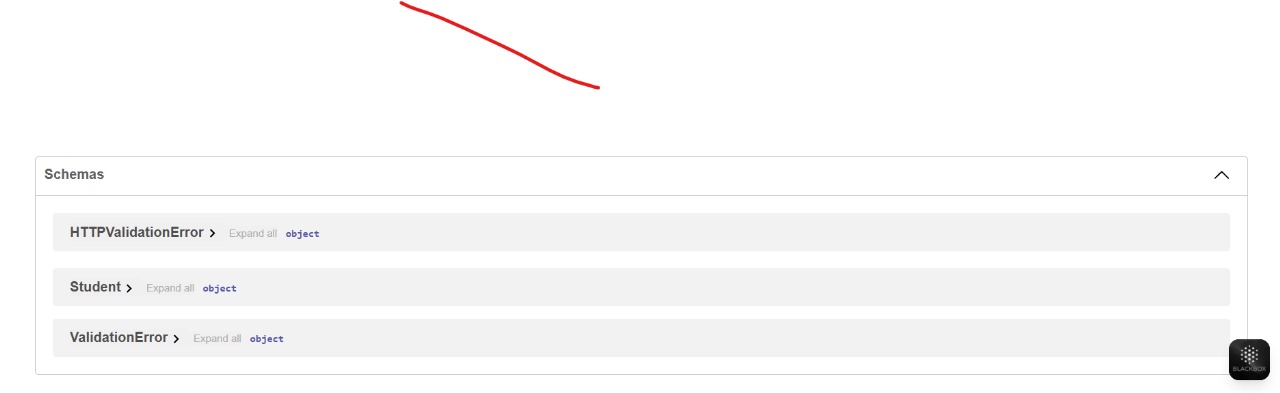
1. Start the server



1. Install winget python



**OUTPUT**

****

****

**Security Considerations**

Currently, the API does not implement authentication or authorization. In real-world deployments, you should:

✅ Use **OAuth2** or **JWT** for user authentication  
✅ Implement **rate limiting** to prevent abuse  
✅ Use **HTTPS** for secure data transfer  
✅ Validate data thoroughly to avoid injection attacks

**Deployment Options**

This FastAPI application can be deployed in various ways:

✅ **Local development**: Using Uvicorn (e.g., uvicorn main:app --reload)  
✅ **Docker containerization**: For portability  
✅ **Cloud platforms**:

* **AWS EC2 / Lambda**
* **Heroku**
* **Azure App Service**
* **Google Cloud Run**

Each deployment option has its pros and cons regarding cost, scalability, and ease of use.

**Further Research and Enhancements**

To make this project more robust and production-ready, consider:

1)Replace in-memory list with a database (PostgreSQL, MongoDB, SQLite).  
 2)Use **SQLAlchemy** or **Tortoise ORM** for database interactions.  
 3)Add user authentication (OAuth2, JWT).  
 4)Implement **pagination** for large datasets.  
 5)Write **unit tests** using **pytest** and **FastAPI’s TestClient**.  
 6)Add caching (Redis) for frequently requested data.  
 7)Create CI/CD pipelines for automated testing and deployment.  
 8)Add API versioning to manage changes over time.

**Conclusion**

This project provides a solid foundation for building APIs with FastAPI. It demonstrates how to create a CRUD API, validate data, and leverage asynchronous programming for performance. FastAPI’s built-in interactive docs (Swagger UI and ReDoc) also make it easy for developers and consumers to understand and use the API.

With further enhancements like database integration, authentication, and testing, this simple CRUD API can evolve into a fully-featured, production-ready service.