Python Project 15: Blog API

1. Project Overview

Build a simple **Blog REST API** using FastAPI and SQLite.

The API allows clients to:

- Create blog posts
- View all posts
- View one post
- Update a post
- Delete a post
- Search posts

FastAPI uses Pydantic models to validate request bodies and can automatically generate interactive API documentation.

2. Features

- REST API
- Create posts
- Read posts
- Update posts
- Delete posts
- Search posts
- SQLite database
- Request validation
- Response models
- Automatic Swagger documentation
- No frontend required

3. Technologies

- Python 3
- FastAPI
- Pydantic
- SQLite
- Uvicorn
- sqlite3

FastAPI response models validate and document returned data and can filter fields that should not be exposed.

4. Folder Structure

```
blog_api/
□□□ main.py
□□□ database.py
□□□ schemas.py
□□□ requirements.txt
□□□ README.md
```

5. How the Project Works

```
Client
 □
HTTP Request
 □
FastAPI
 □
Validate Request
 □
Database Functions
 □
SQLite
 □
```

JSON Response

The main HTTP operations are:

```
POST    /posts
GET     /posts
GET     /posts/{post_id}
PUT     /posts/{post_id}
DELETE  /posts/{post_id}
GET     /posts/search/{keyword}
```

6. Complete Database Code

database.py

```
import sqlite3


class BlogDatabase:

    def __init__(self, database_name="blog.db"):
        self.database_name = database_name
        self.create_table()

    def connect(self):
        return sqlite3.connect(self.database_name)

    def create_table(self):
        connection = self.connect()
        cursor = connection.cursor()

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS posts (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                title TEXT NOT NULL,
```

```
            content TEXT NOT NULL,
            author TEXT NOT NULL
        )
    """)

    connection.commit()
    connection.close()

def create_post(self, title, content, author):
    connection = self.connect()
    cursor = connection.cursor()

    cursor.execute("""
        INSERT INTO posts
        (title, content, author)
        VALUES (?, ?, ?)
    """, (title, content, author))

    connection.commit()

    post_id = cursor.lastrowid

    connection.close()

    return post_id

def get_posts(self):
    connection = self.connect()
    connection.row_factory = sqlite3.Row

    cursor = connection.cursor()

    cursor.execute("""
        SELECT id, title, content, author
```

```
            FROM posts
            ORDER BY id DESC
        """)

        posts = [dict(row) for row in cursor.fetchall()]

        connection.close()

        return posts

    def get_post(self, post_id):
        connection = self.connect()
        connection.row_factory = sqlite3.Row

        cursor = connection.cursor()

        cursor.execute("""
            SELECT id, title, content, author
            FROM posts
            WHERE id = ?
        """, (post_id,))

        post = cursor.fetchone()

        connection.close()

        return dict(post) if post else None

    def update_post(
        self,
        post_id,
        title,
        content,
        author
```

```
    ):
        connection = self.connect()
        cursor = connection.cursor()

        cursor.execute("""
            UPDATE posts
            SET title = ?,
                content = ?,
                author = ?
            WHERE id = ?
        """, (
            title,
            content,
            author,
            post_id
        ))

        connection.commit()

        updated = cursor.rowcount > 0

        connection.close()

        return updated

    def delete_post(self, post_id):
        connection = self.connect()
        cursor = connection.cursor()

        cursor.execute("""
            DELETE FROM posts
            WHERE id = ?
        """, (post_id,))
```

```
        connection.commit()

        deleted = cursor.rowcount > 0

        connection.close()

        return deleted

    def search_posts(self, keyword):
        connection = self.connect()
        connection.row_factory = sqlite3.Row

        cursor = connection.cursor()

        pattern = f"%{keyword}%"

        cursor.execute("""
            SELECT id, title, content, author
            FROM posts
            WHERE title LIKE ?
               OR content LIKE ?
               OR author LIKE ?
            ORDER BY id DESC
        """, (
            pattern,
            pattern,
            pattern
        ))

        posts = [dict(row) for row in cursor.fetchall()]

        connection.close()

        return posts
```

The database uses parameterized SQL placeholders instead of building SQL queries from strings, following Python's SQLite guidance.

7. Complete Schema Code

schemas.py

```
from pydantic import BaseModel, Field


class PostCreate(BaseModel):
    title: str = Field(
        min_length=1,
        max_length=200
    )

    content: str = Field(
        min_length=1
    )

    author: str = Field(
        min_length=1,
        max_length=100
    )


class Post(PostCreate):
    id: int
```

Pydantic models allow FastAPI to validate incoming JSON request bodies automatically.

8. Complete API Code

main.py

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

```
from database import BlogDatabase
from schemas import Post, PostCreate


app = FastAPI(
    title="Blog API",
    description="A simple REST API for managing blog posts.",
    version="1.0.0"
)

database = BlogDatabase()


@app.get("/")
def home():
    return {
        "message": "Blog API is running"
    }


@app.post(
    "/posts",
    response_model=Post,
    status_code=201
)
def create_post(post: PostCreate):

    post_id = database.create_post(
        post.title,
        post.content,
        post.author
    )
```

```
    return {
        "id": post_id,
        **post.model_dump()
    }


@app.get(
    "/posts",
    response_model=list[Post]
)
def get_posts():

    return database.get_posts()


@app.get(
    "/posts/{post_id}",
    response_model=Post
)
def get_post(post_id: int):

    post = database.get_post(post_id)

    if post is None:
        raise HTTPException(
            status_code=404,
            detail="Post not found."
        )

    return post


@app.put(
    "/posts/{post_id}",
```

```
    response_model=Post
)
def update_post(
    post_id: int,
    post: PostCreate
):

    updated = database.update_post(
        post_id,
        post.title,
        post.content,
        post.author
    )

    if not updated:
        raise HTTPException(
            status_code=404,
            detail="Post not found."
        )

    return {
        "id": post_id,
        **post.model_dump()
    }


@app.delete("/posts/{post_id}")
def delete_post(post_id: int):

    deleted = database.delete_post(post_id)

    if not deleted:
        raise HTTPException(
            status_code=404,
```

```
            detail="Post not found."
        )

    return {
        "message": "Post deleted successfully."
    }


@app.get(
    "/posts/search/{keyword}",
    response_model=list[Post]
)
def search_posts(keyword: str):

    return database.search_posts(keyword)
```

9. Requirements

requirements.txt

```
fastapi
uvicorn
```

10. README

README.md

```
# Blog API

A simple REST API for creating and managing blog posts.

## Features

- Create posts
- Get all posts
```

- Get one post
- Update posts
- Delete posts
- Search posts
- SQLite database
- Automatic API documentation

Installation

Create a virtual environment:

```
python -m venv venv
```

Activate it and install dependencies:

```
pip install -r requirements.txt
```

Run

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

API Documentation

Open:

```
http://127.0.0.1:8000/docs
```

FastAPI automatically provides interactive Swagger UI documentation at `/docs` when the application is running.

11. How Frontend / Client + Backend Connect

There is no dedicated GUI frontend in this project. The API itself is the backend.

A frontend, mobile app, Postman, or another client can communicate with it through HTTP:

```
Client
  ↓
POST /posts
  ↓
FastAPI
  ↓
Pydantic validation
  ↓
SQLite
  ↓
JSON response
```

Example request:

```
{
   "title": "Learning FastAPI",
   "content": "FastAPI makes building APIs with Python simple.",
   "author": "Hafsa"
}
```

Example response:

```
{
   "id": 1,
   "title": "Learning FastAPI",
   "content": "FastAPI makes building APIs with Python simple.",
   "author": "Hafsa"
}
```

12. How to Run

Open the project directory:

```
cd blog_api
```

Create a virtual environment:

```
python -m venv venv
```

Activate it and install the dependencies:

```
pip install -r requirements.txt
```

Start the server:

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

Open:

```
http://127.0.0.1:8000/docs
```

Use Swagger UI to test every endpoint.

13. Basic Testing

Create Post

```
POST /posts
```

```
{
   "title": "Python APIs",
   "content": "Building APIs is an important backend skill.",
   "author": "Hafsa"
}
```

Get All Posts

```
GET /posts
```

Get One Post

`GET /posts/1`

Update Post

`PUT /posts/1`

```
{
   "title": "Learning FastAPI",
   "content": "FastAPI provides validation and automatic documentation.",
   "author": "Hafsa"
}
```

Search

`GET /posts/search/Python`

Delete

`DELETE /posts/1`

14. Expected API Status Codes

```
201   Post created
200   Request successful
404   Post not found
422   Invalid request data
```

FastAPI uses the declared Pydantic models to validate request data and generate the corresponding API schema.

15. Small Improvements

- Add user authentication
- Add categories and tags

- Add comments
- Add pagination
- Add post timestamps
- Add likes
- Add image uploads
- Add JWT authentication
- Add PostgreSQL
- Add a React frontend
- Deploy the API online

Visit haas.dev for more resources and guides.