FastAPI: The Modern Standard

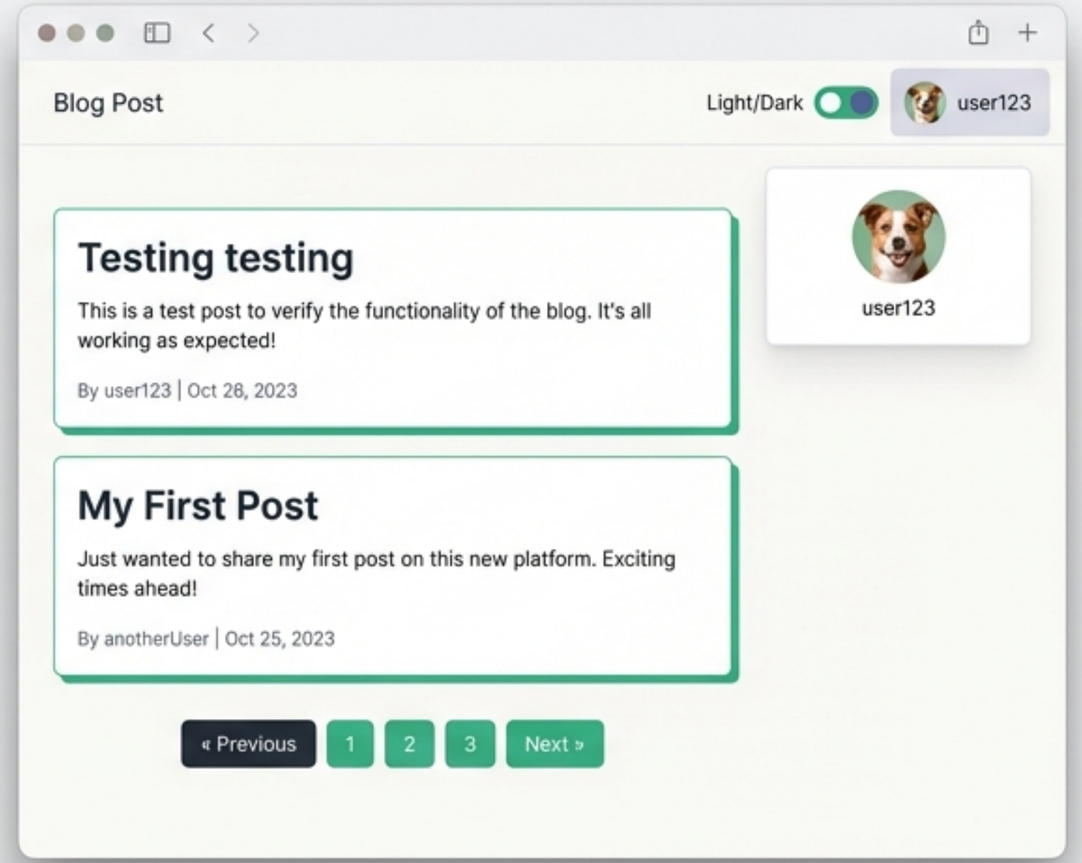
Part 1: From Zero to REST API

Based on the tutorial by Corey Schafer

The Goal: A Full-Stack Blog Application

We are building a production-grade web app featuring:

- Back-end: `JSON REST API`
- Front-end: `HTML with Jinja2`
- Auth: `JWT & Password Hashing`
- Database: `SQLAlchemy (SQLite -> Postgres)`
- Background Tasks: `Email resets`



Why FastAPI?

High Performance

On par with NodeJS and Go. Powered by Starlette and Pydantic.

Automatic Documentation

Generates interactive Swagger UI docs without extra config.

Modern Python

Built on Python 3.6+ standard type hints for fewer bugs.

Async Support

Native handling of asynchronous code (async/await).

The Setup: Modern Package Management

The Modern Way (Recommended)

```
$ uv init fastAPI_blog
$ uv add "fastapi[standard]"
```

Uses 'uv' for blazing fast dependency management.

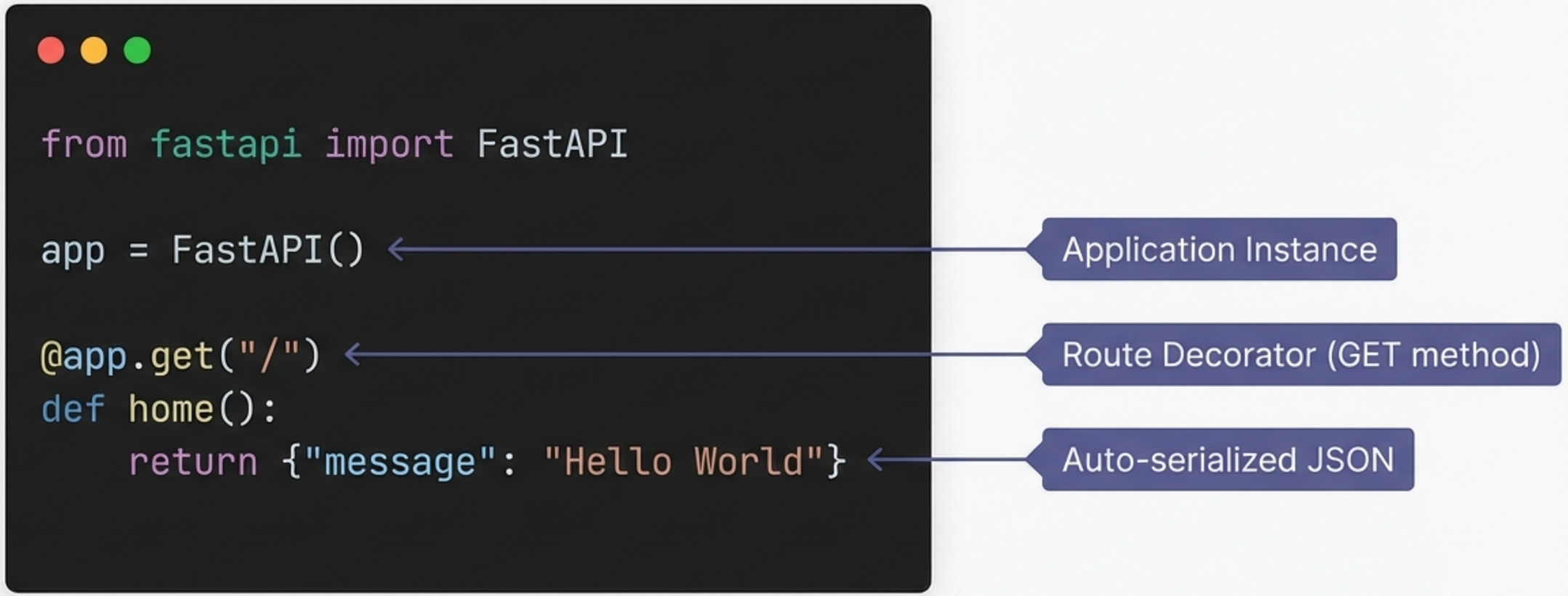
The Standard Way

```
$ mkdir fastAPI_blog
$ pip install "fastapi[standard]"
```

Standard pip installation.

Installing 'fastapi[standard]' includes the framework, uvicorn server, and CLI tools.

Five Lines to a Working Server



Technical Editorial

Running the Application

Development Mode

```
$ fastapi dev main.py
INFO: Will watch for changes in these directories: [...]
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process...
```

Enables auto-reload. Server restarts on code save.

Production Mode

```
$ fastapi run main.py
```

Optimized for performance. Auto-reload disabled.

If using uv: “uv run fastapi dev main.py”

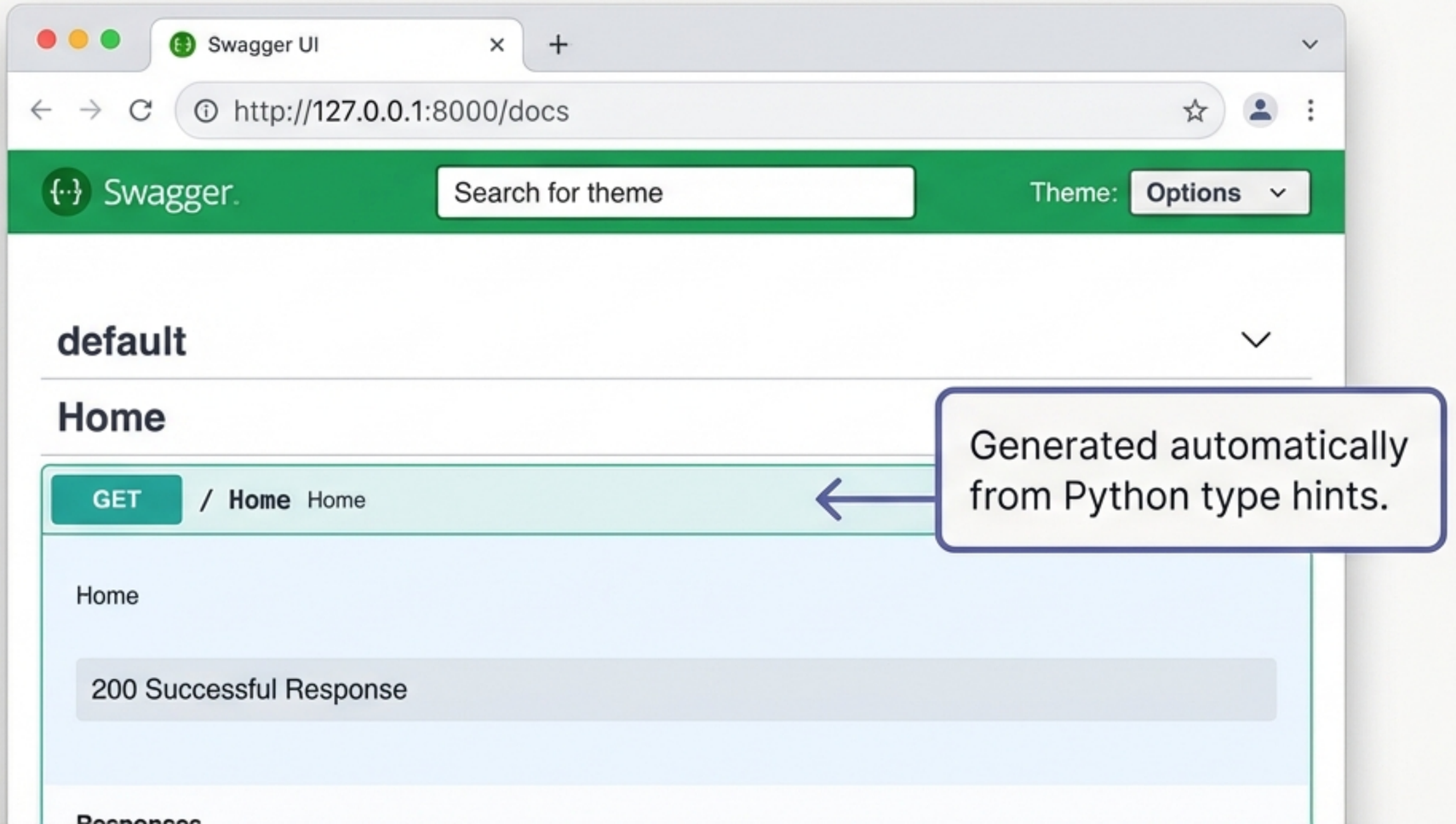
Technical Editorial

The Result: Automatic JSON Serialization

- FastAPI automatically infers content type.
- Returning a Python dictionary = Valid JSON response.
- No manual serialization (jsonify) required.

Technical Editorial

The Magic: Zero-Config Documentation

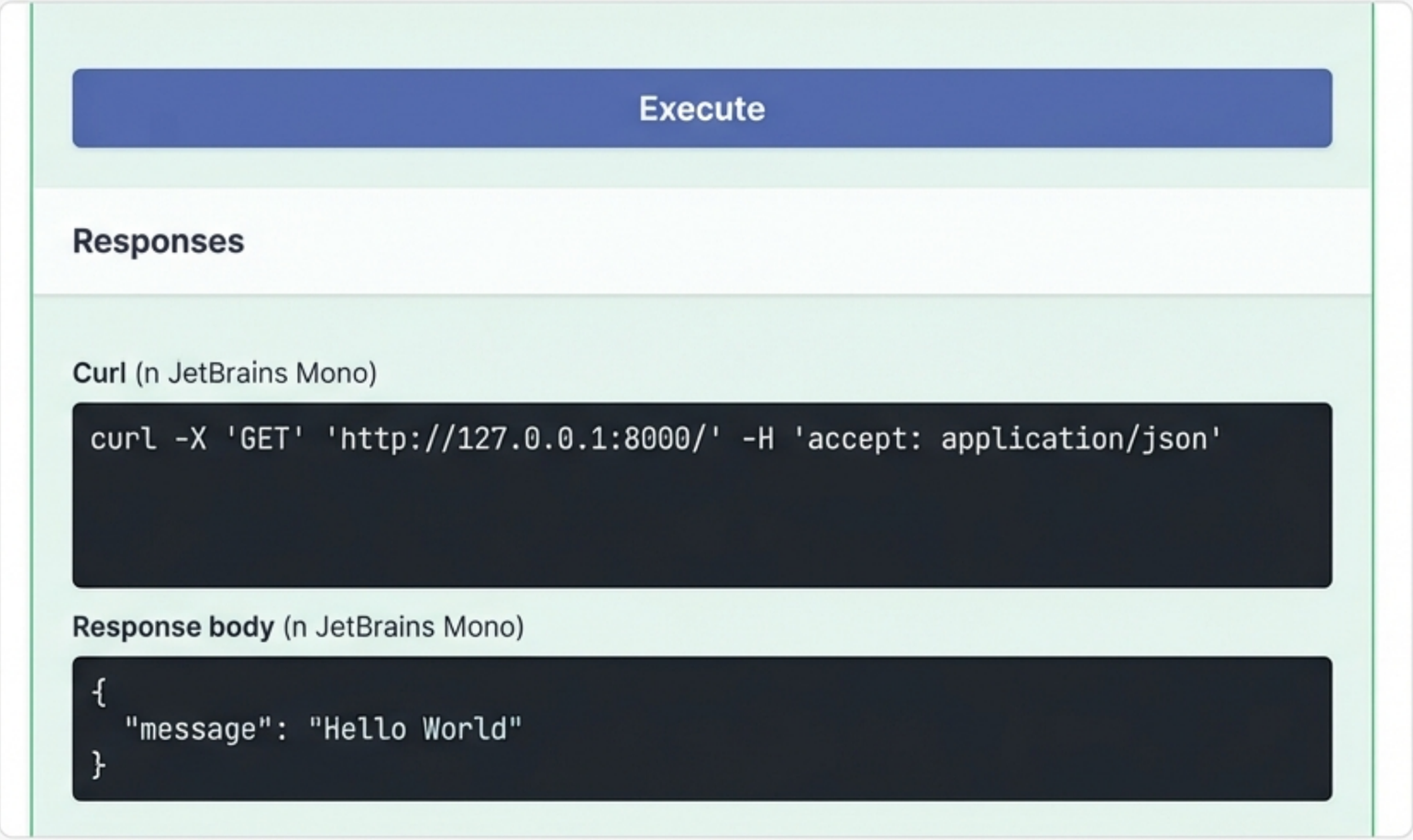


We didn't write a single line of documentation.

FastAPI inspects the code to build an OpenAPI schema.

Also available: /redoc

Interactive Testing & Curl Generation



The docs are also a client.

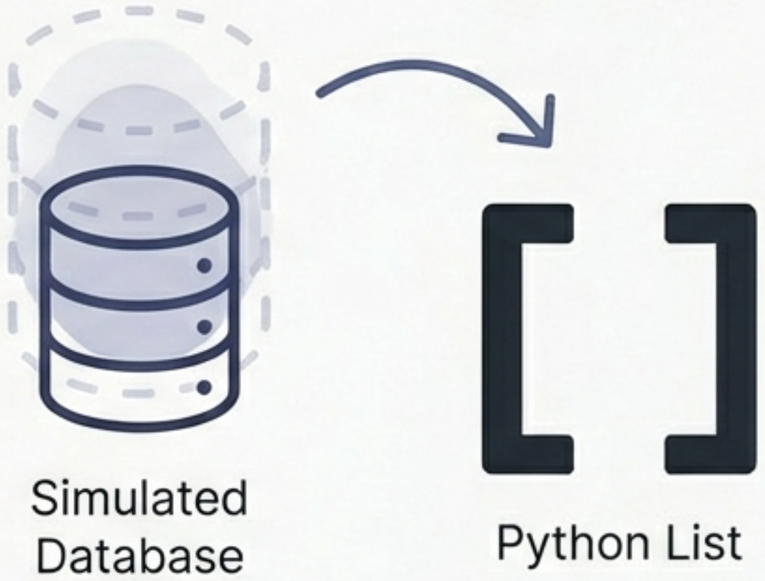
1. Execute requests directly from the browser.
2. Copy the auto-generated Curl command for terminal debugging.

Technical Editorial

Mocking the Database

For Part 1, we simulate a database using a global list of dictionaries.

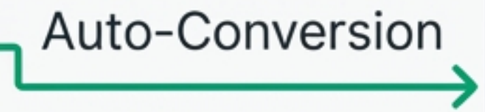
```
my_posts = [
    {"id": 1, "title": "Post 1", "content": "Content 1"},
    {"id": 2, "title": "Post 2", "content": "Content 2"}
]
```



Endpoint 2: The Data Route

The Code

```python
@app.get("/api/posts")
def get_posts():
    return my_posts
```



The Output

```json
[
  {
    "id": 1,
    "title": "Post 1",
    "content": "Content 1"
  },
  {
    "id": 2,
    "title": "Post 2",
    "content": "Content 2"
  }
]
```

Route is prefixed with /api to distinguish from HTML views.

Technical Editorial

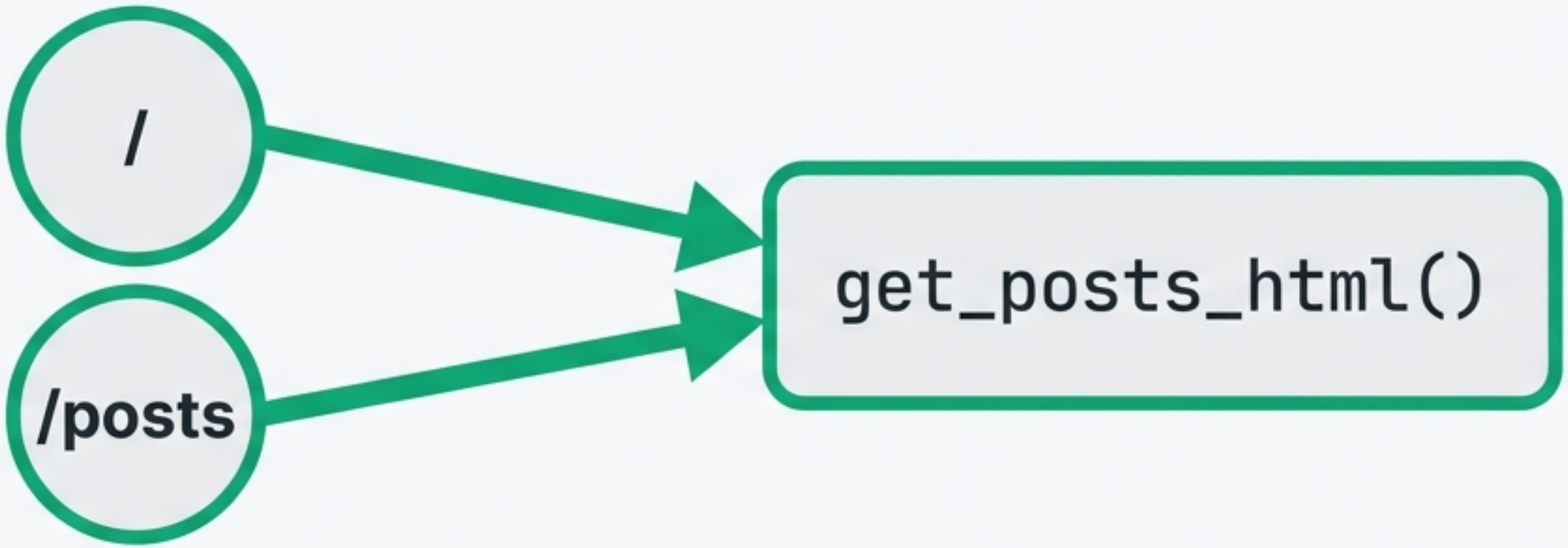
Serving HTML to Humans

FastAPI can serve raw HTML, though templates are preferred for larger apps.

```
from fastapi.responses import HTMLResponse

@app.get("/", response_class=HTMLResponse)
def home():
    return f"<h1>{my_posts[0]['title']}</h1>"
```

Advanced Routing: Stacked Decorators




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

Multiple URLs can trigger the same function by stacking decorators.

Technical Editorial

Cleaning the Schema

- **Problem:** HTML routes clutter the API documentation intended for developers.
- **Solution:** Use ‘include_in_schema=False’.

```
@app.get("/posts", response_class=HTMLResponse, include_in_schema=False)
def get_posts_html():
    ...
```

GET /api/posts

~~GET /posts~~

Summary & Next Steps

What We Built

- Configured environment with ‘uv’
- Created a JSON API (`/api/posts`)
- Created HTML routes (`/posts`)
- Explored Auto-Docs (Swagger UI)
- Cleaned schema with `include_in_schema`

Coming Next in Part 2

- Templates: Replacing raw strings with `Jinja2`
- Styling: CSS inheritance and layout
- Databases: Moving from mock lists to SQLAlchemy

Next