Flask
web development,
one drop at a time

FastAPI

MODULE 10

DESIGNING AND IMPLEMENTING RESTFUL APIS IN PYTHON USING FLASK AND FASTAPI

Introduction to RESTful APIs

> Architectural style for designing networked applications.

> Key Principles
  – Stateless communication,
  – uniform interface,
  – resource-based,
  – client-server architecture.

> Used for communication between web servers and clients.

> Enables modular and scalable architecture.

> Supports communication between diverse systems and platforms.

Representational State Transfer (REST)

> The concept essentially has two parts

  – The resources and how we identify them

  – The way we operate or work with these resources.

> REST was described in 2000 by Roy Thomas Fielding in a paper called '*Architectural Styles and the Design of Network-based Software Architectures*'.

> It describes how to work with resources using the HTTP protocol and the features offered by this protocol.

REST in Five Steps

> A Short Primer

1. Give everything an ID
2. Link things together
3. Use standard HTTP methods
4. Support multiple representations
5. Use stateless communications

REST in Five Steps

> Give Everything an ID

- Use URIs for entity IDs

```
http://example.com/userDirectory/user/johnDoe
http://example.com/userDirectory/user/{login}
http://example.com/userDirectory/users
```

REST in Five Steps

> Link Things Together

- Query

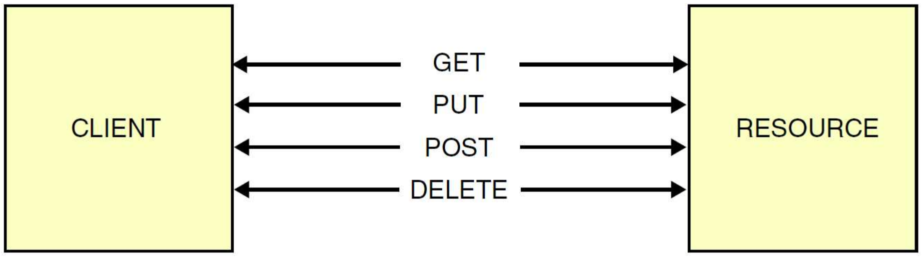
```
http://example.com/userDirectory/users
```

- Response

```
<customers>
    <customer ref="/userDirectory/user/johnDoe"/>
    <customer ref="/userDirectory/user/janeDoe"/>
    <customer ref="/userDirectory/user/jimKirk"/>
</customers>
```

REST in Five Steps

> Use Standard HTTP Methods



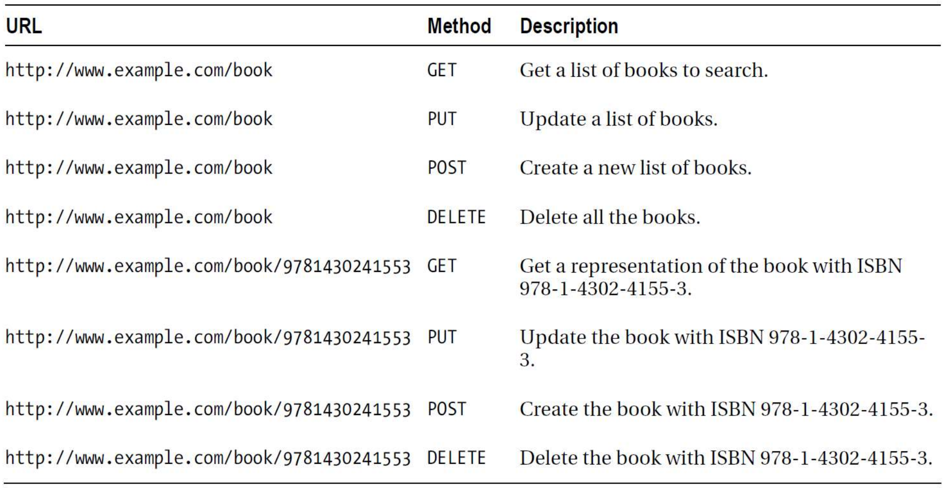
REST in Five Steps

> Use Standard HTTP Methods

– Leverage standard semantics for HTTP methods

Method	Purpose
GET	Read, possibly cached
POST	Update or create without a known ID
PUT	Update or create with a known ID
DELETE	Remove

Example



URL	Method	Description
`http://www.example.com/book`	GET	Get a list of books to search.
`http://www.example.com/book`	PUT	Update a list of books.
`http://www.example.com/book`	POST	Create a new list of books.
`http://www.example.com/book`	DELETE	Delete all the books.
`http://www.example.com/book/9781430241553`	GET	Get a representation of the book with ISBN 978-1-4302-4155-3.
`http://www.example.com/book/9781430241553`	PUT	Update the book with ISBN 978-1-4302-4155-3.
`http://www.example.com/book/9781430241553`	POST	Create the book with ISBN 978-1-4302-4155-3.
`http://www.example.com/book/9781430241553`	DELETE	Delete the book with ISBN 978-1-4302-4155-3.

REST in Five Steps

> Support Multiple Representations

- Offer data in a variety of formats
  - XML
  - JSON
  - (X)HTML

> Support content negotiation

- Accept header

```
GET /foo
Accept: application/json
```

- URI-based

```
GET /foo.json
```

REST in Five Steps

> Use Stateless Communications

- Long lived identifiers
- Avoid sessions
- Everything required to process a request contained in the request

HATEOAS Principle

> Hypermedia as the Engine of Application State (HATEOAS)

> RESTful applications should offer a single, fixed entry point URL.

> All related resources should be:

- Dynamically discovered: Resources are discovered dynamically through provided hypermedia links.
- Dynamically navigated: Navigation across resources starts from the fixed entry point and uses hypermedia links.

> Hypermedia links should be consistently utilized across all resources to guide interactions and state transitions.

Common Patterns: Container, Item

- Server in Control of URI Path Space
  - List container contents: GET /container
  - Add item to container: POST /container
    - With item in request
    - URI of item returned in HTTP response header

      e.g. Location: http://host/container/item
- Read item: GET /container/item
- Update item: PUT /container/item
  - With updated item in request
- Remove item: DELETE /container/item

13

Common Patterns: Map, Key, Value

- Client in Control of URI Path Space
  - List key-value pairs: GET /map
  - Put new value to map: PUT /map/{key}
    - With entry in request

      e.g. PUT /map/dir/contents.xml
  - Read value: GET /map/{key}
  - Update value: PUT /map/{key}
    - With updated value in request
  - Remove value: DELETE /map/{key}

14

Advantages of the RESTful Approach

- **Minimized Client-Side Errors**: Simplifies client implementation, reducing coding mistakes.
- **Prevention of Invalid State Transitions**: Ensures proper interaction with system states.
- **Backward Compatibility**: Enables gradual updates without breaking older client versions.
- **Addressability**: Clear and consistent resource identification via URLs.
- **Standardized Interface**: Unified operations through HTTP methods
- **Protocol Stability**: Relies on the robust and widely adopted HTTP protocol.
- **Interoperability**: Promotes seamless communication across diverse platforms.
- **Widespread Adoption**: Supported by numerous tools/libraries/frameworks.
- **User Familiarity**: Leverages widely known web standards for ease of use.

Key Benefits

- **Server-Side**
  - **Horizontal Scalability**: Easily accommodates increased load by adding more servers.
  - **Simplified Failover**: Ensures reliability and quick recovery in case of server issues.
  - **Cacheability**: Enhances performance through efficient caching mechanisms.
  - **Reduced Coupling**: Promotes flexibility and easier system maintenance.
  - **Seamless Integration**: Operates efficiently with existing web infrastructure.

Key Benefits

> **Client-Side**

- **Bookmarkable Resources**: Easy to save and revisit specific states or pages.
- **Browser-Friendly**: Supports quick experimentation and testing directly in a web browser.
- **Multi-Language Compatibility**: Broad support across various programming languages.
- **Flexible Data Formats**: Offers the flexibility to choose among formats like JSON, XML, etc

Drawbacks of REST

> If the system is a very large one, then designing based on REST could become a very complex task.

- No direct bridge to the OOP world
- No standard formal language to describe interaction, like WSDL

> Restrictions for GET length sometimes may be a problem.

> Implementing Security on a REST system is an issue.

Getting Started with Flask

> Installing Flask

- Make sure you have Python installed on your system
- `pip –version`
- `pip install Flask`
- `flask --version`

Flask Overview

> Microframework Philosophy

- Flask is often described as a microframework
- Its design philosophy is centered around simplicity, minimalism, and modularity

Simplicity and Minimalism

> Lightweight Core

- Flask has a small and concise core.
- It provides just the essentials needed to build web applications without unnecessary features, making it easy to learn and use.

Simplicity and Minimalism

> No Opinionated Components

- Flask doesn't impose a specific way of doing things.
- It gives developers the flexibility to choose their preferred tools and libraries for various tasks, such as database integration, form handling, or authentication.

Simplicity and Minimalism

> Explicit is Better Than Implicit

- Flask follows the principle that explicit code is more readable and maintainable.
- Developers have clear control over their application structure and components.

Modularity

- > Choose Only What You Need: Flask is designed with a modular architecture, allowing developers to pick and choose the components they need for their specific project. This modularity contributes to a leaner and more efficient application.

Modularity

- > Extensibility
  - – Flask is easily extensible through the use of extensions.
  - – Developers can add functionality to their applications by integrating specific extensions for tasks like database connectivity, authentication, and more.

Modularity

> Extensibility

- Flask is easily extensible through the use of extensions.
- Developers can add functionality to their applications by integrating specific extensions for tasks like database connectivity, authentication, and more.

Flexibility

> Flexibility Over Convention

- Unlike some frameworks that follow the convention over configuration principle, Flask emphasizes flexibility.
- Developers have the freedom to structure their code and define their application's architecture based on their preferences.

Flexibility

> No Built-in ORM (Object-Relational Mapping)

- Flask does not come with an integrated ORM, allowing developers to choose the database abstraction layer or ORM that best fits their needs.
- SQLAlchemy is a popular choice for Flask projects, but developers can opt for others if they prefer.

DIY (Do It Yourself) Mentality

> Empowering Developers

- Flask adopts a DIY mentality, empowering developers to take control of their application's structure and components.
- This philosophy aligns with the Pythonic principle of giving developers the tools to do their work effectively without unnecessary constraints.

DIY (Do It Yourself) Mentality

> Encouraging Creativity

  - Flask encourages developers to be creative and innovative in solving problems.
  - It's not prescriptive in its approach, allowing developers to implement solutions that make sense for their specific use cases.

Hello World Endpoint: Creating a Basic Endpoint

```
from flask import Flask

# Create a Flask application instance
app = Flask(__name__)

# Define a route for the root URL ("/")
@app.route('/')
def hello_world():
    return 'Hello, World!'

# Run the Flask application if this script is executed
if __name__ == '__main__':
    app.run(debug=True)
```

Flask routing

> Flask routing is a fundamental concept that allows you to map URLs (Uniform Resource Locators) to specific functions in your web application.

> Routing defines how the application responds to different HTTP requests on different URLs.

> In Flask, routing is achieved using decorators on Python functions.

33

Example

```
import json
from flask import Flask, jsonify, Response
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['world']
countries1 = db.countries1

app = Flask(__name__)

@app.route('/world/api/v1.0/countries/<code>', methods=['GET'])
def countryByCode(code):
    return jsonify(countries1.find_one({"_id": code}))

@app.route('/world/api/v1.0/countries', methods=['GET'])
def countries():
    return json.dumps([e for e in countries1.find({})])

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

34

Multiple Routes for a Single Function

> A single function can handle multiple routes by applying multiple @app.route decorators:

```
@app.route('/')
@app.route('/home')
def home():
    return 'Welcome to the home page!'
```

Redirects and Errors

> Flask allows you to perform redirects and handle errors using the redirect and abort functions:

```
from flask import redirect, abort

@app.route('/redirect_example')
def redirect_example():
    return redirect('/new_location')

@app.route('/error_example')
def error_example():
    abort(404)  # Raises a 404 error
```

Static Files

> Flask automatically serves static files (like CSS or images) from a folder named static in the application's root directory.

> For example, a file `style.css` in the static folder would be accessible at `/static/style.css`.

URL Building

> Flask provides the **`url_for`** function to build URLs dynamically based on the function name:

```
from flask import url_for

@app.route('/')
def home():
    return f'The URL for home is {url_for("home")}'
```

> This ensures that changes to URLs are automatically reflected throughout your application.

Testing Flask APIs

```
import pytest
from your_app import create_app

@pytest.fixture
def app():
    app = create_app(testing=True)
    yield app

@pytest.fixture
def client(app):
    return app.test_client()
```

Testing Flask APIs

```
def test_hello_world(client):
    response = client.get('/')
    assert response.status_code == 200
    assert b'Hello, World!' in response.data
```

Documentation with Swagger/OpenAPI

> Swagger is a set of open-source tools for designing, building, and documenting RESTful APIs.

> OpenAPI, formerly known as Swagger Specification, is a standard for describing RESTful APIs.

> Combining Flask with Swagger/OpenAPI allows developers to automatically generate interactive and comprehensive API documentation.

Flask-RESTPlus

```
from flask import Flask
from flask_restplus import Api, Resource

app = Flask(__name__)
api = Api(app, version='1.0', title='My API', description='An API example')

@api.route('/hello')
class HelloWorld(Resource):
    def get(self):
        """Returns 'Hello, World!'"""
        return {'message': 'Hello, World!'}

if __name__ == '__main__':
    app.run(debug=True)
```

Best Practices

- Follow to best practices ensures maintainability, scalability, and overall code quality.
- Organize by Feature
  - Structure your project based on features or modules rather than strictly adhering to a particular pattern.
  - Group related functionality together.

Best Practices

- Blueprints
  - Use Flask Blueprints to modularize your application.
  - Blueprints allow you to define sets of routes in separate files and then register them with the main application.

Best Practices

```
from flask import Blueprint
from flask_restplus import Api

api_bp = Blueprint('api', __name__)
api = Api(api_bp, version='1.0', title='My API', description='A sample API')

ns = api.namespace('tasks', description='Task operations')

@ns.route('/')
class TaskList(Resource):
    def get(self):
        return tasks

    def post(self):
        task = api.payload
        tasks.append(task)
        return task, 201
```

Best Practices

> Separation of Concerns

- Follow the principle of separation of concerns.
- Keep your business logic separate from presentation and configuration concerns.

Best Practices

> Use Configuration Files

- Store configuration settings in separate configuration files, and use different configurations for development, testing, and production environments.

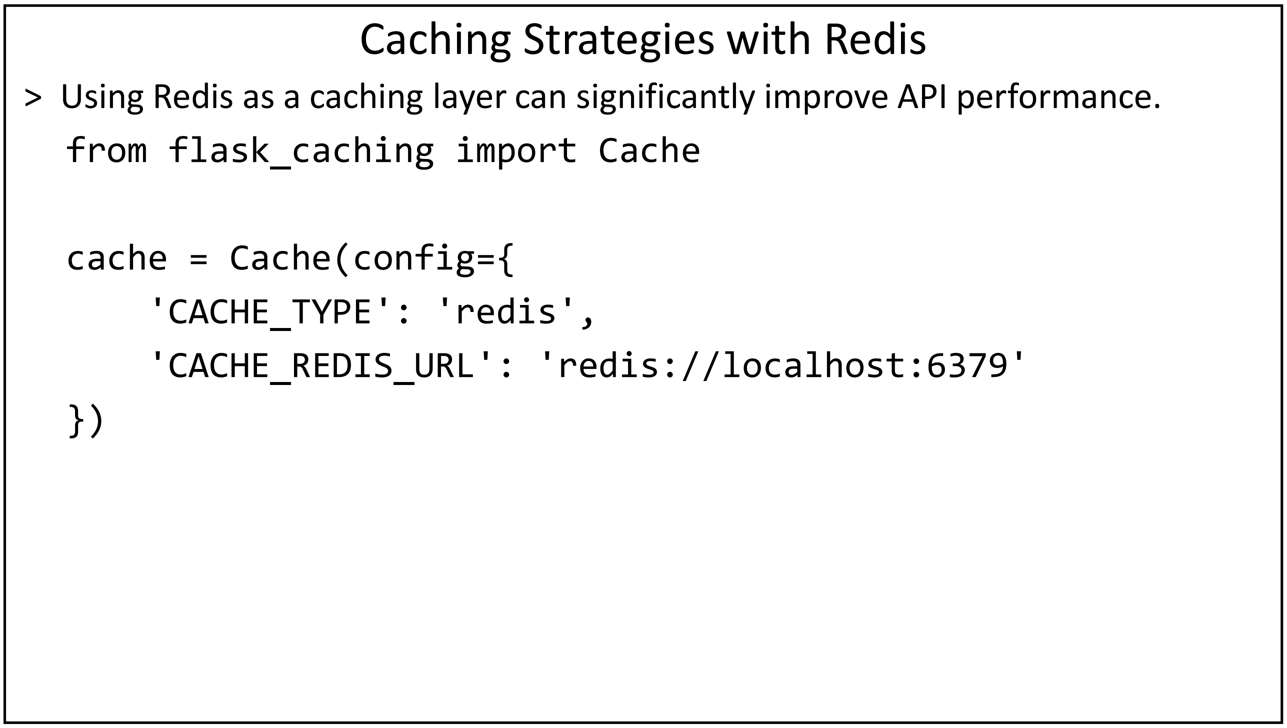
> Environment Variables

- For sensitive information (like secret keys), use environment variables rather than hardcoding them in the code.

Best Practices

```
from config import config

def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    return app
```

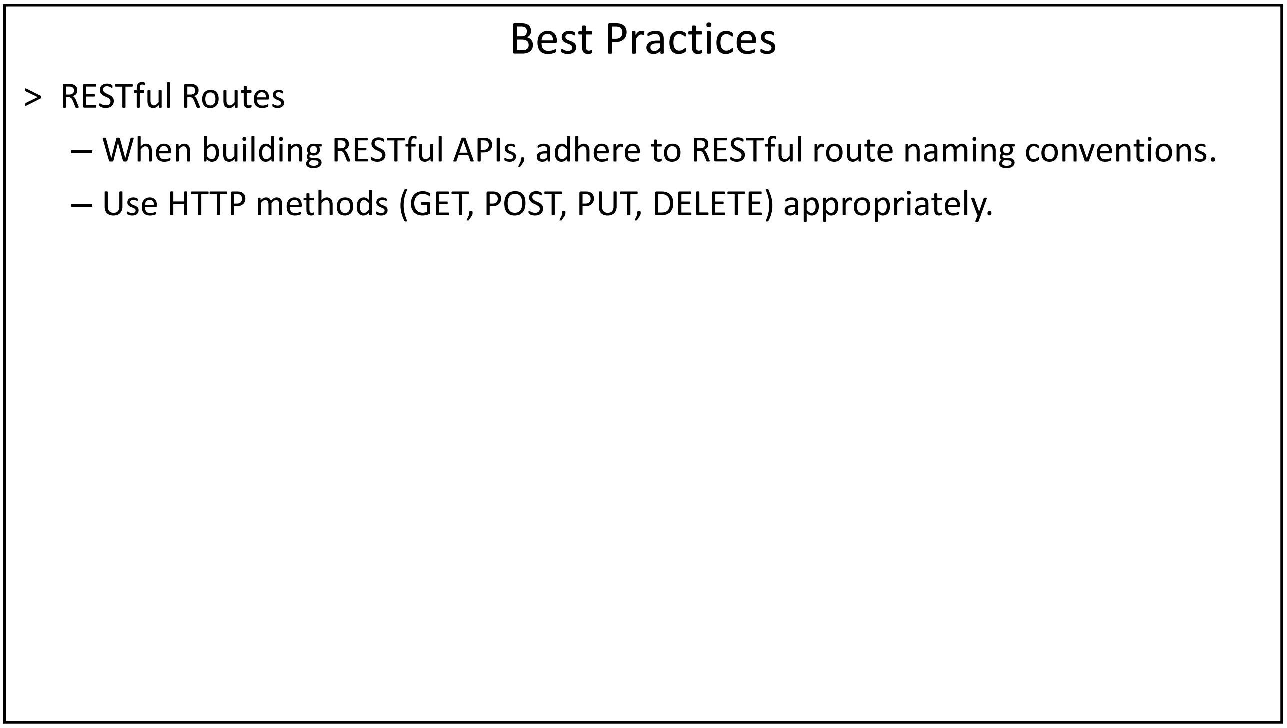


Caching Strategies with Redis

> Using Redis as a caching layer can significantly improve API performance.

```
from flask_caching import Cache

cache = Cache(config={
    'CACHE_TYPE': 'redis',
    'CACHE_REDIS_URL': 'redis://localhost:6379'
})
```



Best Practices

> RESTful Routes

- When building RESTful APIs, adhere to RESTful route naming conventions.
- Use HTTP methods (GET, POST, PUT, DELETE) appropriately.

Best Practices

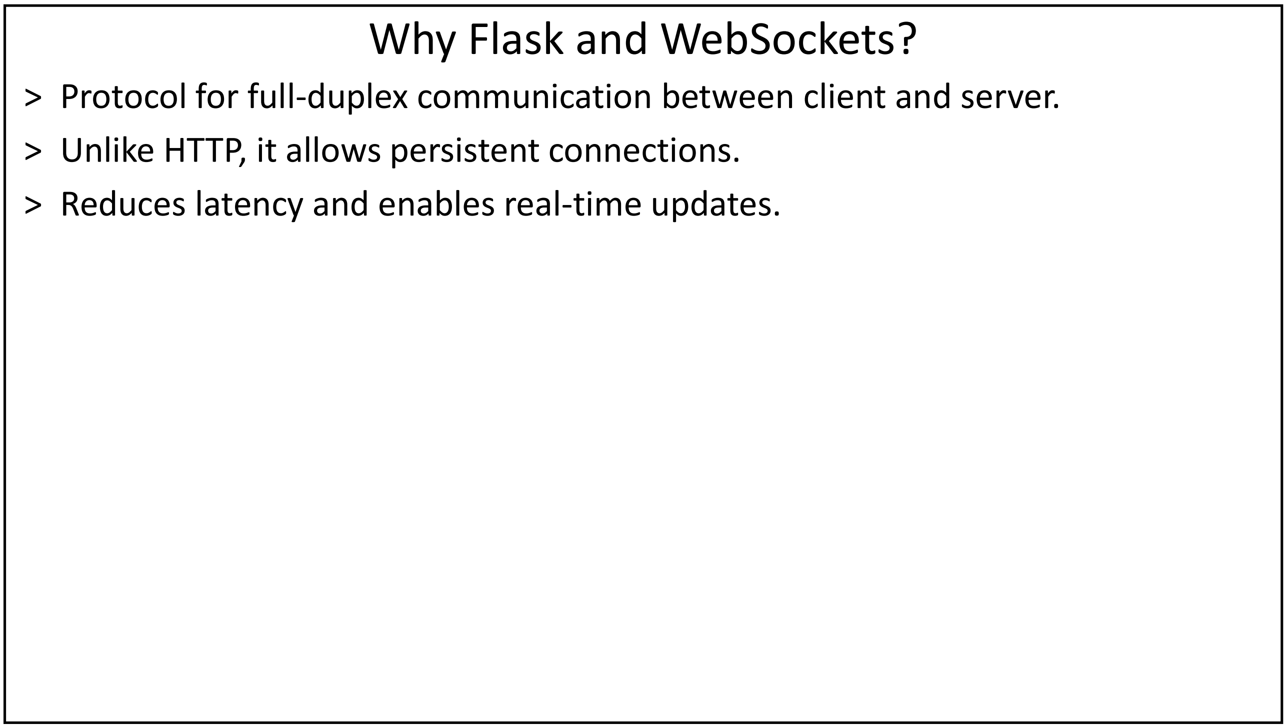
- Single Responsibility
  - Keep views (functions associated with routes) simple and focused on a single responsibility.
  - Consider breaking down large views into smaller functions or methods.

Secure Your API with JWT

- JSON Web Tokens (JWT) offer a method for protecting your API endpoints. You can use libraries like Flask-JWT-Extended.

```
from flask_jwt_extended import JWTManager

app.config['JWT_SECRET_KEY'] = 'super-secret'
jwt = JWTManager(app)
```



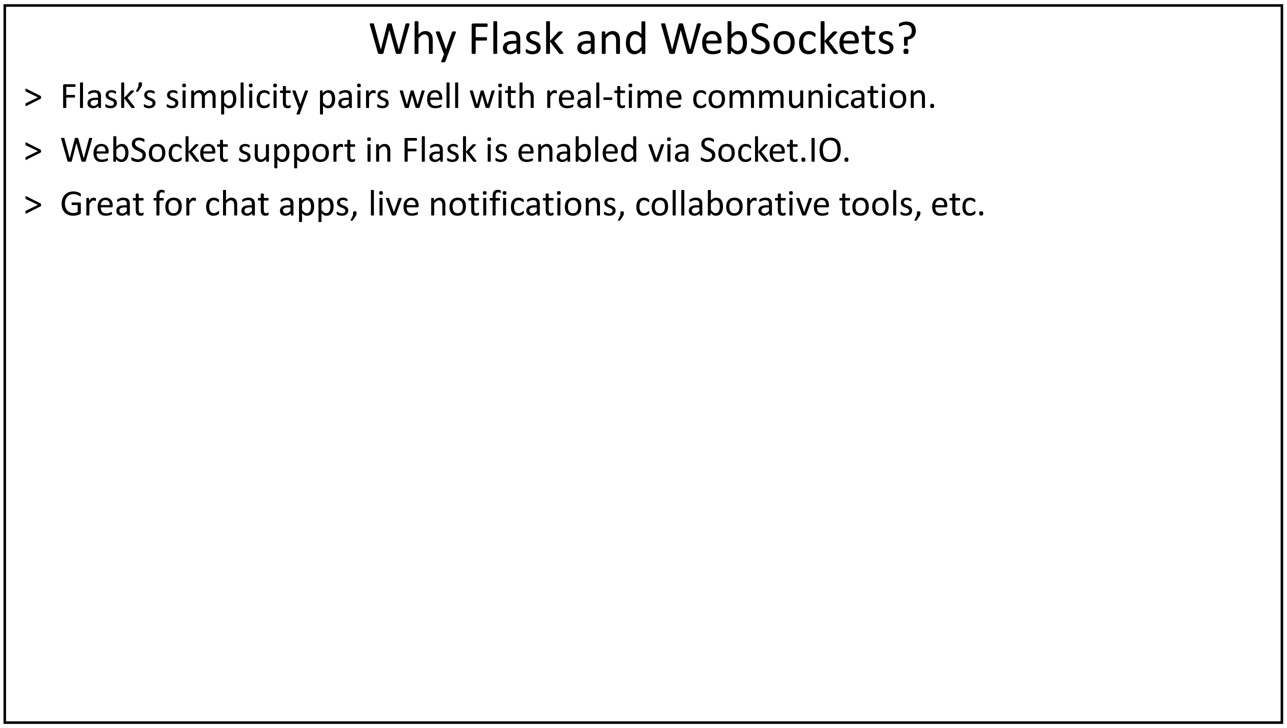
Why Flask and WebSockets?

> Protocol for full-duplex communication between client and server.

> Unlike HTTP, it allows persistent connections.

> Reduces latency and enables real-time updates.

53



Why Flask and WebSockets?

> Flask’s simplicity pairs well with real-time communication.

> WebSocket support in Flask is enabled via Socket.IO.

> Great for chat apps, live notifications, collaborative tools, etc.

54

Introduction to Socket.IO

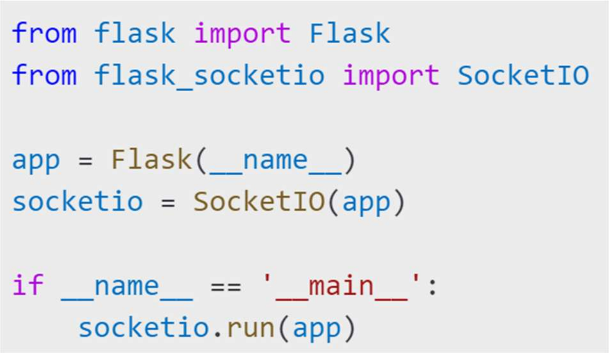
- Library for real-time, bi-directional communication.
- Supports WebSockets and falls back to polling if necessary.
- Offers both client-side (JavaScript) and server-side (Python) implementations.

Setting Up Flask and Socket.IO

- Requirements
  - Python 3.x
  - Flask
  - Flask-SocketIO
- Install via pip

```
pip install flask flask-socketio
```

Integrating Flask-SocketIO



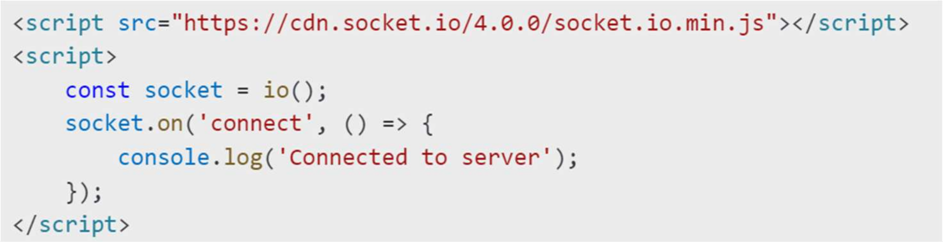
```python
from flask import Flask
from flask_socketio import SocketIO

app = Flask(__name__)
socketio = SocketIO(app)

if __name__ == '__main__':
    socketio.run(app)
```

Establishing Client-Side Connection

> JavaScript Example



```html
<script src="https://cdn.socket.io/4.0.0/socket.io.min.js"></script>
<script>
    const socket = io();
    socket.on('connect', () => {
        console.log('Connected to server');
    });
</script>
```

Handling Events

> Server-Side:

```
@socketio.on('message')
def handle_message(msg):
    print(f'Received message: {msg}')
```

> Cliend-Side:

```
socket.emit('message', 'Hello, Server!');
```

Broadcasting Messages

> Server-Side:

```
@socketio.on('broadcast')
def handle_broadcast(msg):
    socketio.emit('broadcast', msg)
```

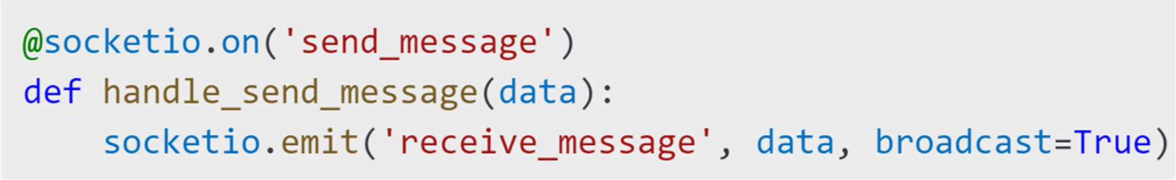
> Cliend-Side:

```
socket.on('broadcast', (msg) => {
    console.log(`Broadcast: ${msg}`);
});
```

Real-Time Chat Application Example

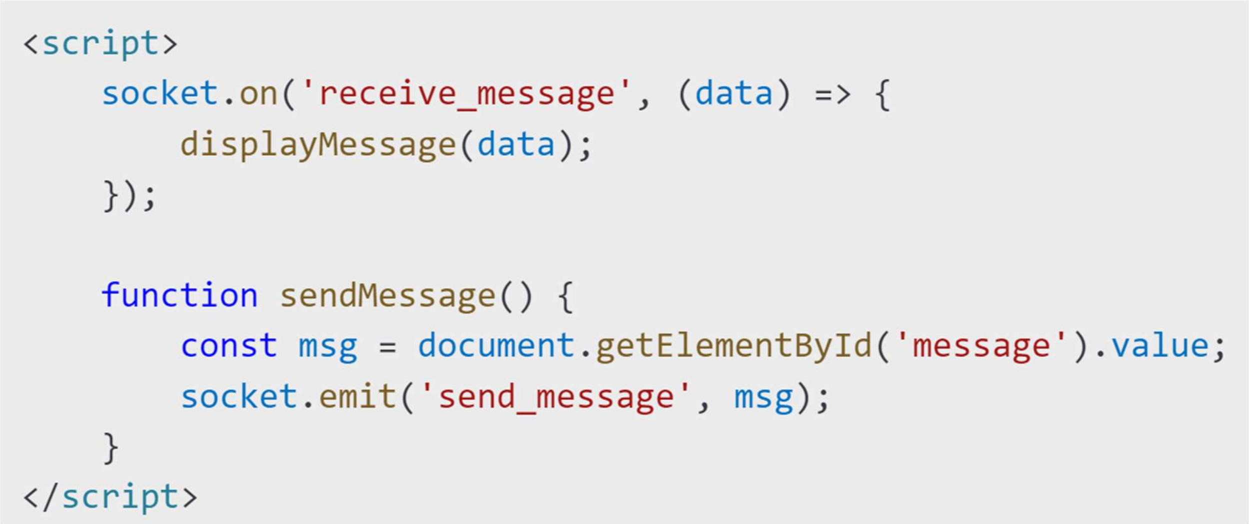
- Flask for the backend.
- Socket.IO for WebSocket communication.
- HTML/JavaScript for the frontend.

Backend Code for Chat App



```python
@socketio.on('send_message')
def handle_send_message(data):
    socketio.emit('receive_message', data, broadcast=True)
```

Frontend Code for Chat App



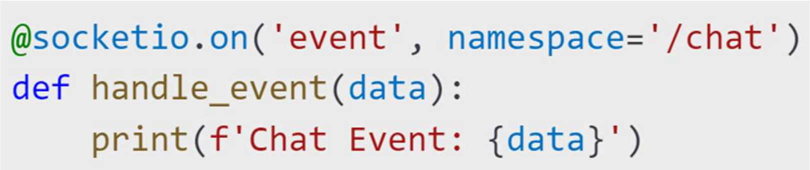
```
<script>
    socket.on('receive_message', (data) => {
        displayMessage(data);
    });

    function sendMessage() {
        const msg = document.getElementById('message').value;
        socket.emit('send_message', msg);
    }
</script>
```

Using Namespaces

> Namespaces allow logical separation of events.

> Server-Side Example:



```
@socketio.on('event', namespace='/chat')
def handle_event(data):
    print(f'Chat Event: {data}')
```

Rooms for Grouping Clients

- > Rooms group users for targeted communication.
- > Server-Side Example:

```
@socketio.on('event', namespace='/chat')
def handle_event(data):
    print(f'Chat Event: {data}')
```

Emitting Events to Specific Rooms

- > Server-Side Example:

```
@socketio.on('send_to_room')
def handle_room_event(data):
    socketio.emit('room_message', data, room=data['room'])
```

Deployment Considerations

- Use a production-grade server like **`Gunicorn`**.
- Enable WebSocket support:

```
gunicorn --worker-class eventlet -w 1 app:app
```

- Configure load balancers for WebSocket traffic

FastAPI

Introduction to FastAPI

> FastAPI is a modern, fast web framework for building APIs with Python 3.6+.

> Key Features:
  - High performance
  - Easy to use
  - Based on standard Python type hints.

69

Why Choose FastAPI?

> Benefits:
  - Fast to code
  - High performance
  - Automatic interactive API documentation
  - Based on OpenAPI and JSON Schema.

70

Core Features

> Key Features:

  - Request validation
  - Dependency injection
  - Asynchronous programming support
  - Automatic documentation with Swagger and ReDoc.

Installation

> Install FastAPI and Uvicorn (ASGI server):

```
pip install fastapi uvicorn
```

Creating Your First API

> 1. Import FastAPI

```
from fastapi import FastAPI
```

Creating Your First API

> 1. Import FastAPI

```
from fastapi import FastAPI
```

> 2. Create an instance

```
app = FastAPI()
```

Creating Your First API

> 1. Import FastAPI

```
from fastapi import FastAPI
```

> 2. Create an instance

```
app = FastAPI()
```

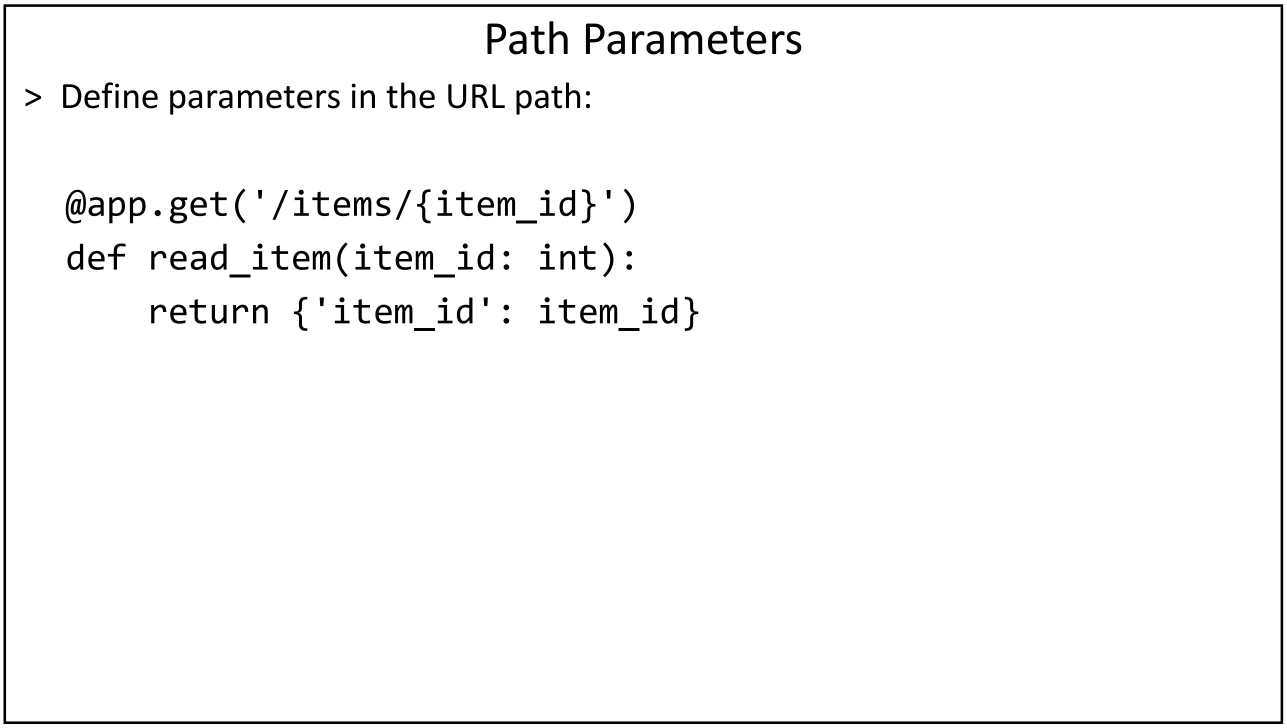
> 3. Define routes

```
@app.get('/')
def read_root():
    return {'Hello': 'World'}
```

Running the API

> Run the app using **Uvicorn**:

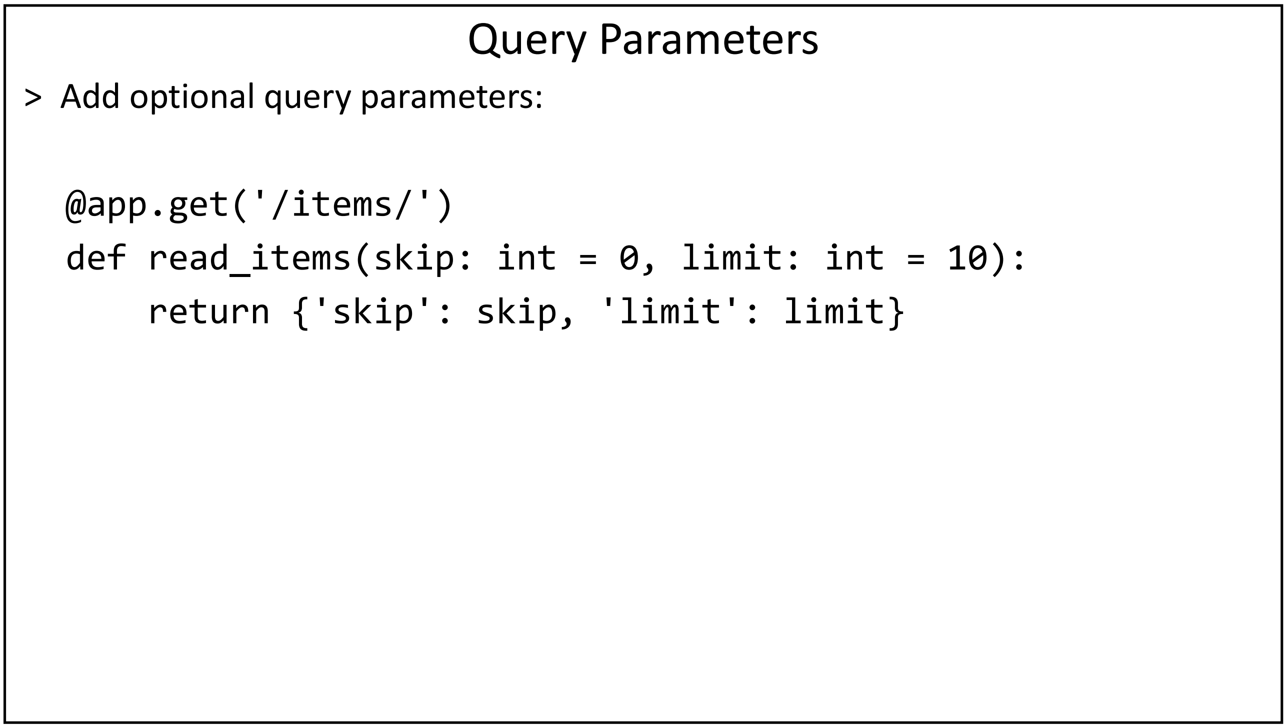
```
uvicorn main:app --reload
```



Path Parameters

> Define parameters in the URL path:

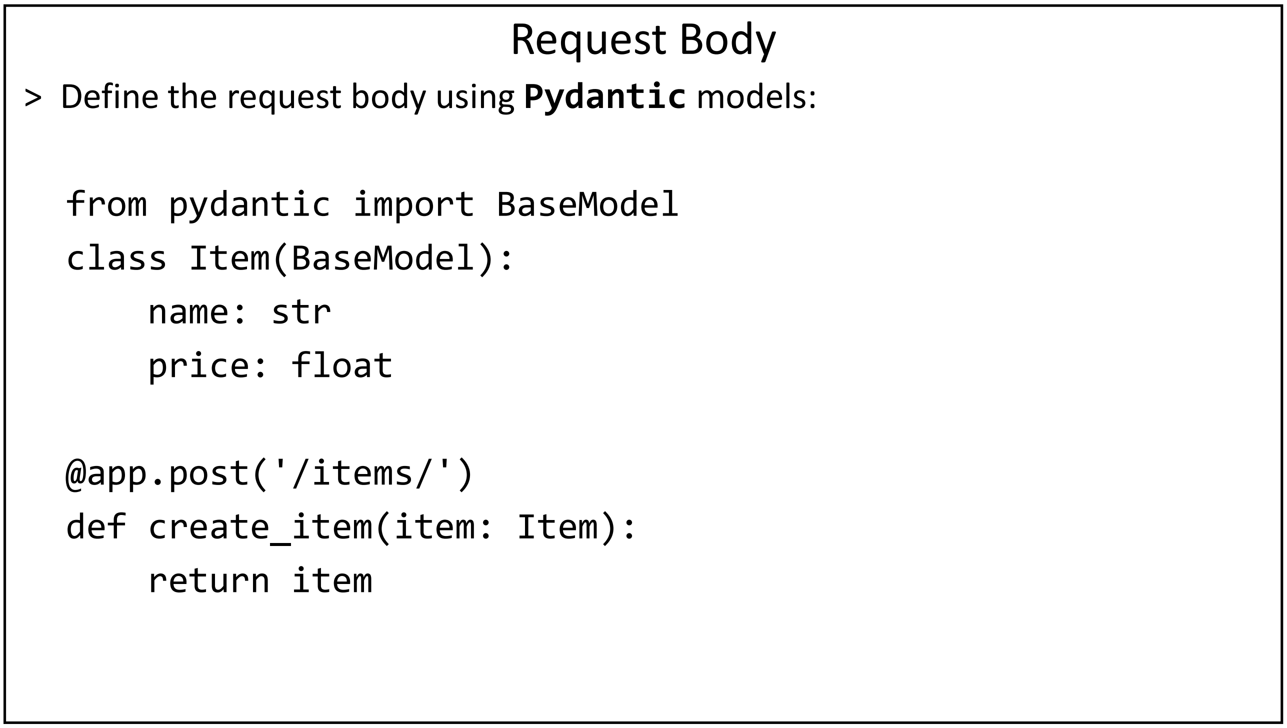
```
@app.get('/items/{item_id}')
def read_item(item_id: int):
    return {'item_id': item_id}
```



Query Parameters

> Add optional query parameters:

```
@app.get('/items/')
def read_items(skip: int = 0, limit: int = 10):
    return {'skip': skip, 'limit': limit}
```



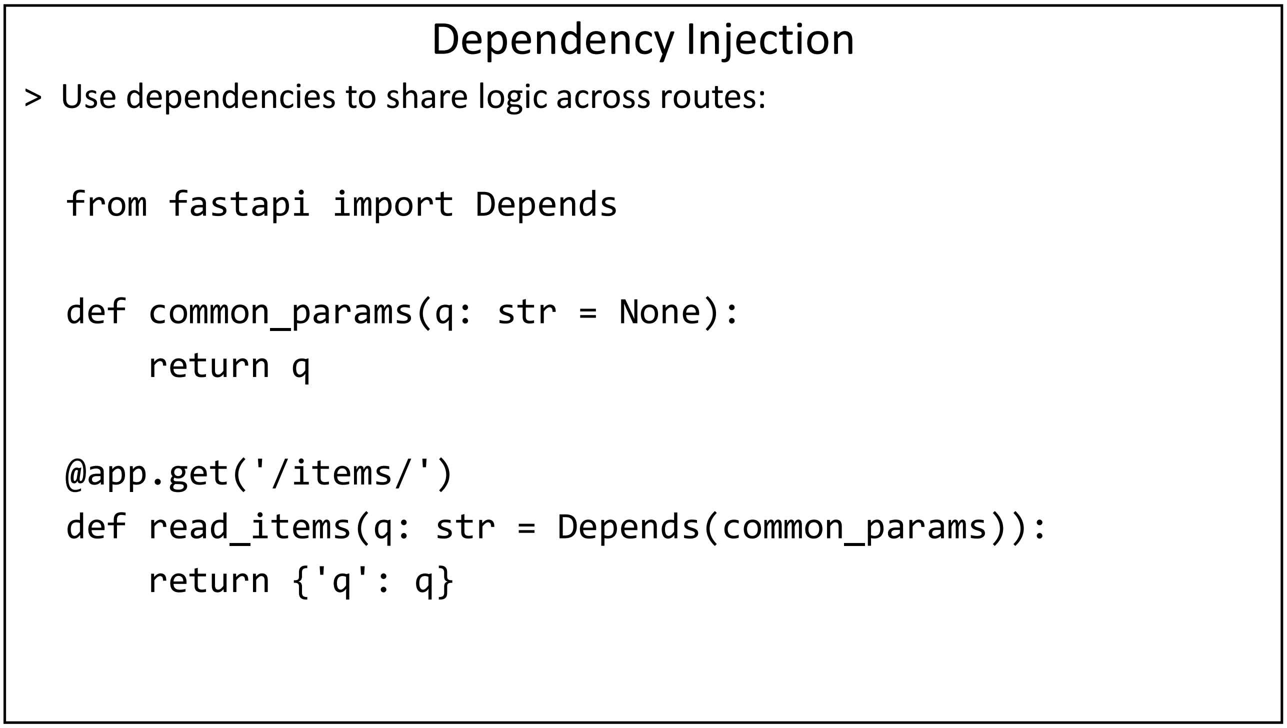
Request Body

> Define the request body using **Pydantic** models:

```
from pydantic import BaseModel
class Item(BaseModel):
    name: str
    price: float

@app.post('/items/')
def create_item(item: Item):
    return item
```

79



Dependency Injection

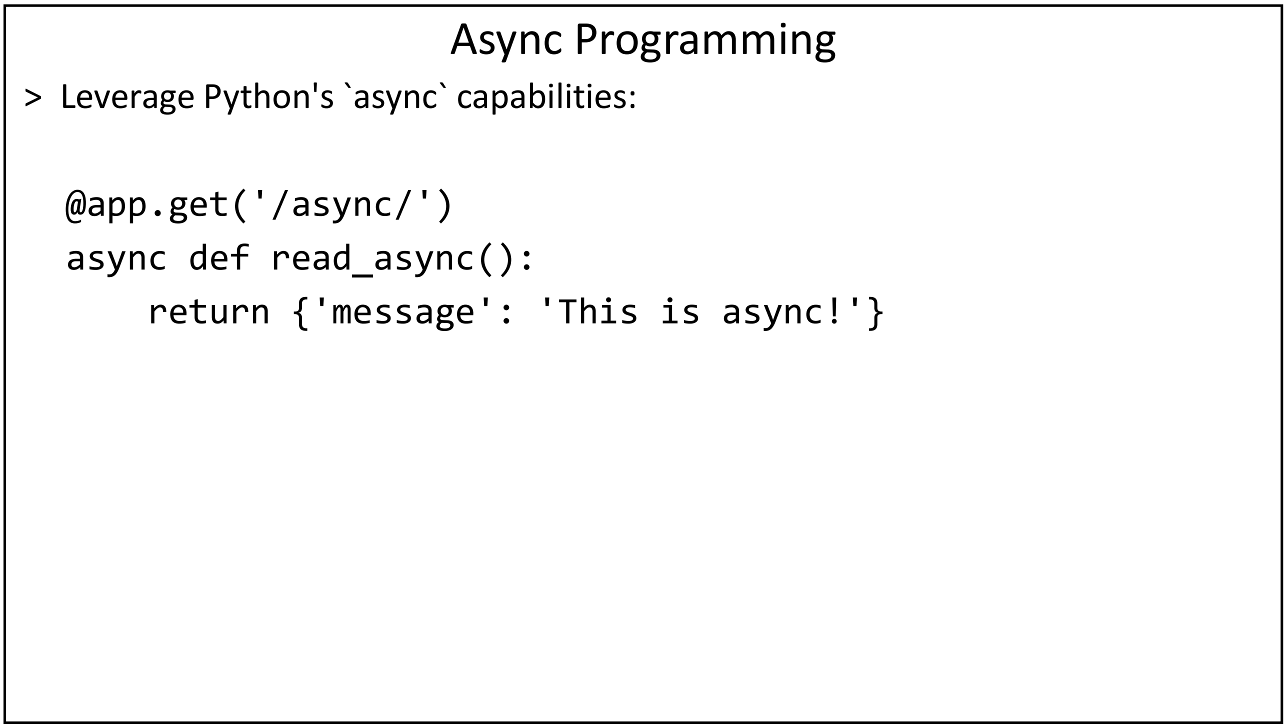
> Use dependencies to share logic across routes:

```
from fastapi import Depends

def common_params(q: str = None):
    return q

@app.get('/items/')
def read_items(q: str = Depends(common_params)):
    return {'q': q}
```

80

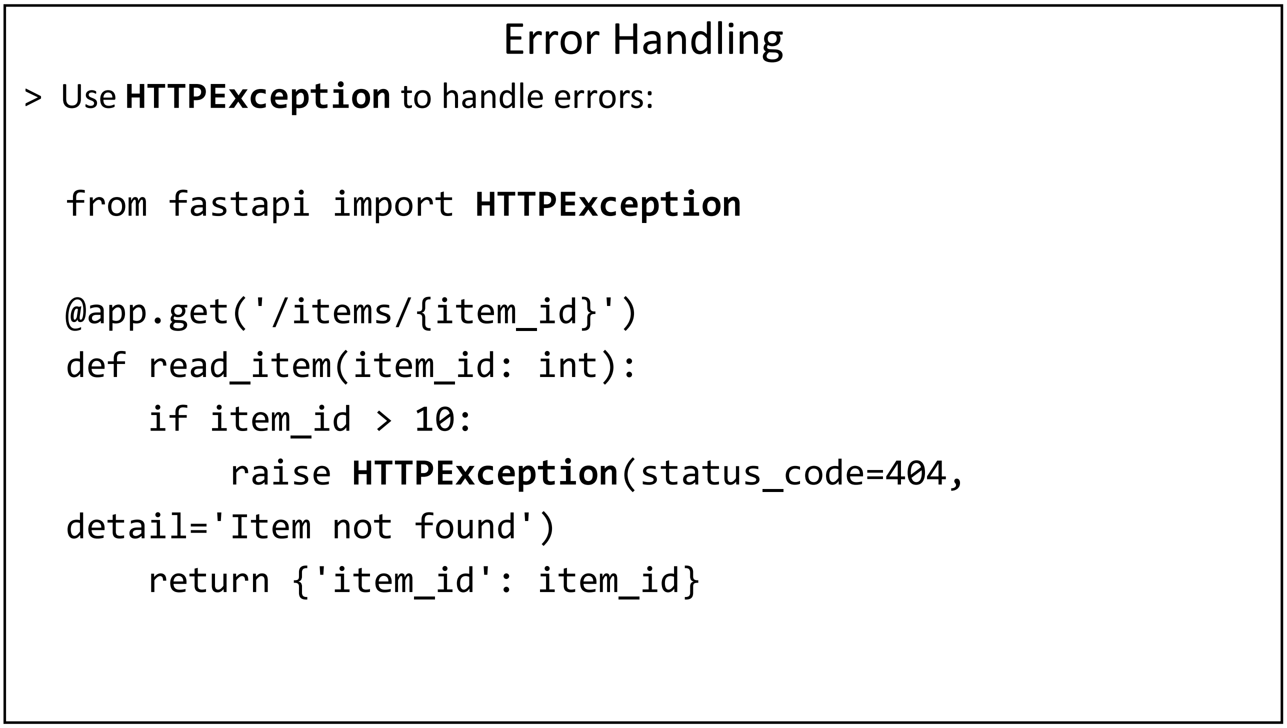


Async Programming

> Leverage Python's `async` capabilities:

```
@app.get('/async/')
async def read_async():
    return {'message': 'This is async!'}
```

81



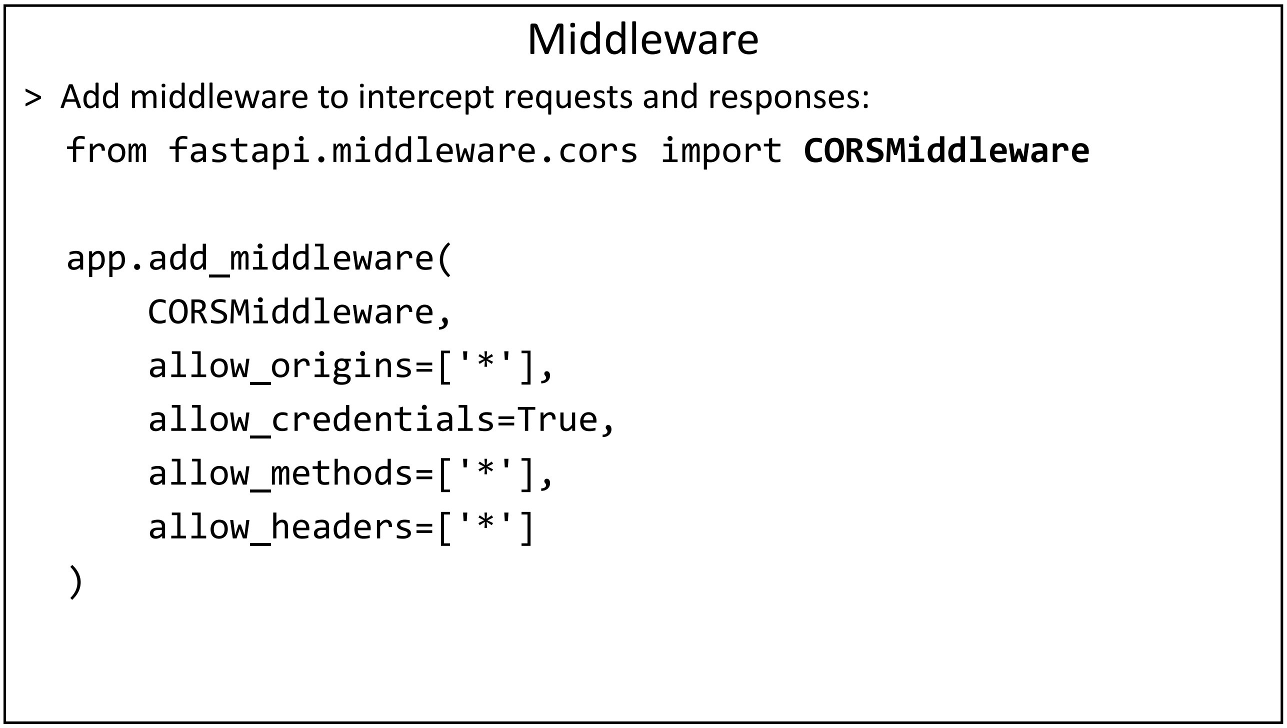
Error Handling

> Use **HTTPException** to handle errors:

```
from fastapi import HTTPException

@app.get('/items/{item_id}')
def read_item(item_id: int):
    if item_id > 10:
        raise HTTPException(status_code=404, 
detail='Item not found')
    return {'item_id': item_id}
```

82



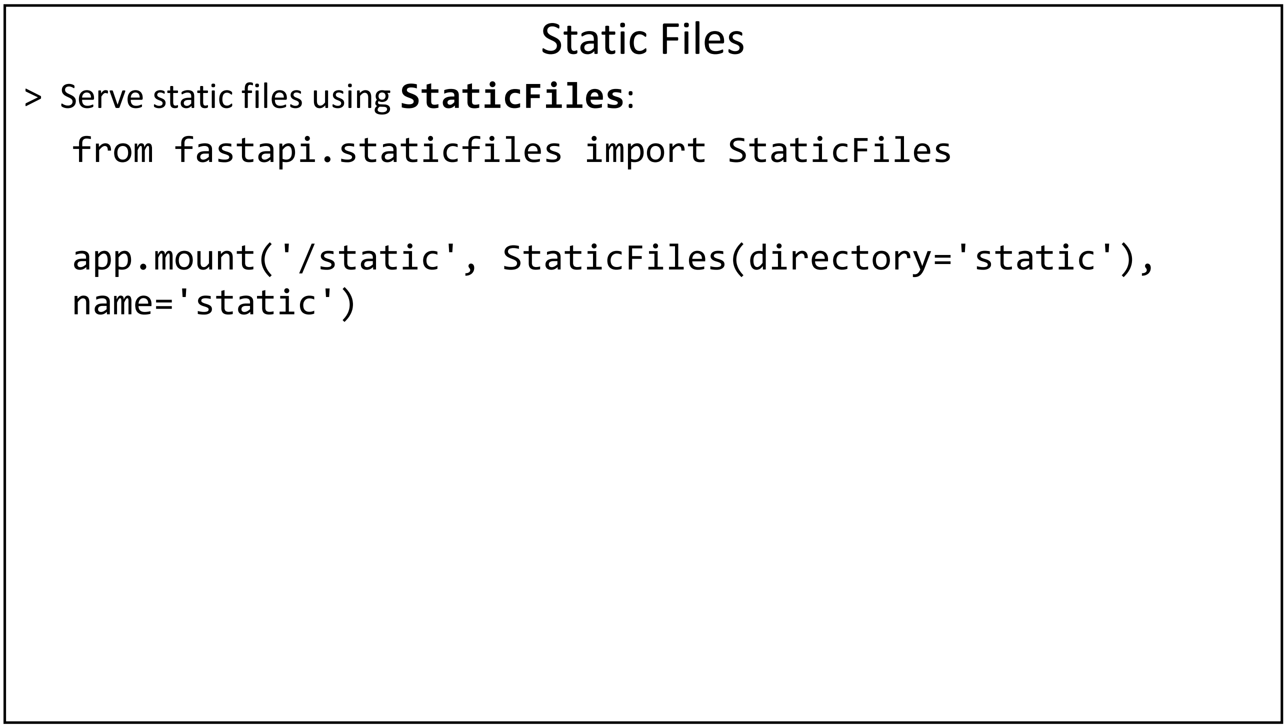
Middleware

> Add middleware to intercept requests and responses:

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=['*'],
    allow_credentials=True,
    allow_methods=['*'],
    allow_headers=['*']
)
```

83



Static Files

> Serve static files using **StaticFiles**:

```
from fastapi.staticfiles import StaticFiles

app.mount('/static', StaticFiles(directory='static'),
name='static')
```

84

Authentication

> FastAPI supports OAuth2 and JWT:

– Use `fastapi.security` for authentication schemes.

> Example:

```
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl='token')
@app.get('/users/me/')
def read_users_me(token: str = Depends(oauth2_scheme)):
    return {'token': token}
```

Swagger UI

> Interactive API docs available by default:

– Swagger UI: http://127.0.0.1:8000/docs

– ReDoc: http://127.0.0.1:8000/redoc

Testing

> FastAPI supports testing with `TestClient`:

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_main():
    response = client.get('/')
    assert response.status_code == 200
    assert response.json() == {'Hello': 'World'}
```

Performance

> FastAPI is built on Starlette and Pydantic:

- Starlette: High-performance ASGI framework.
- Pydantic: Data validation and settings management.

Real-Time Communication with FastAPI WebSockets

A PRACTICAL GUIDE TO IMPLEMENTING WEBSOCKETS WITH FASTAPI

Introduction to WebSockets

> Definition: A communication protocol enabling two-way interactive communication between a client and server over a single TCP connection.

> Key Features

- Full-duplex communication
- Low latency
- Ideal for real-time applications

Why Use WebSockets?

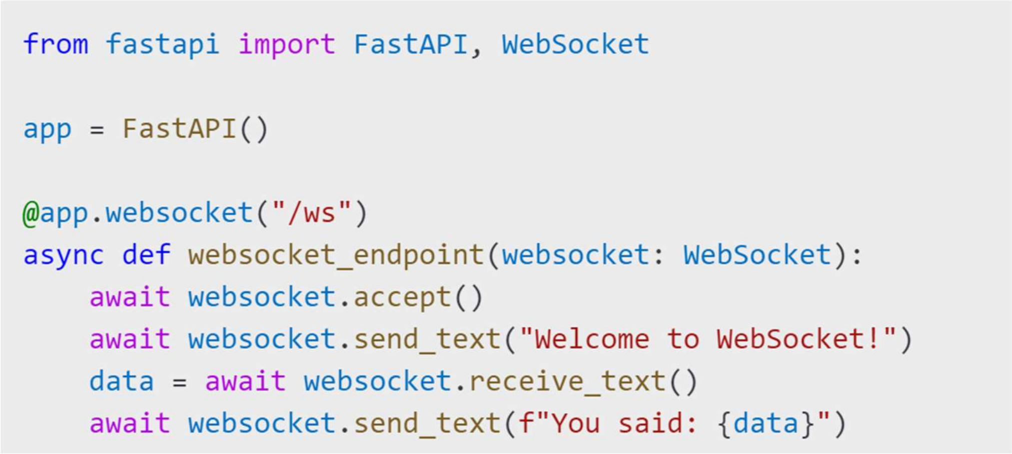
- Real-time data updates (e.g., chat apps, stock prices)
- Interactive applications (e.g., multiplayer games)
- Event-driven communication
- Reduced HTTP overhead compared to REST polling

What Are WebSockets in FastAPI?

- **WebSocket Support**: Built-in support via WebSocket class.
- **Integration**: FastAPI simplifies WebSocket endpoint creation with minimal setup.
- Provides a WebSocket object for managing connections and messages.

Creating a Basic WebSocket Endpoint

> Define a WebSocket route:



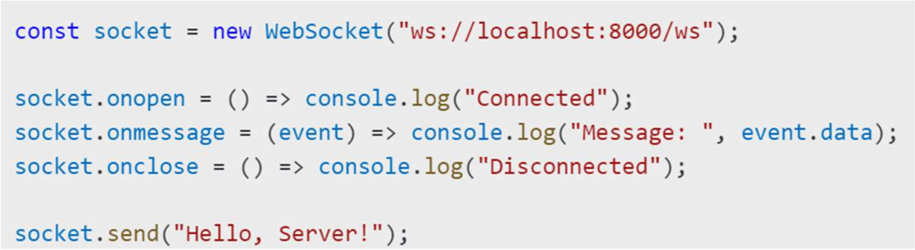
```python
from fastapi import FastAPI, WebSocket

app = FastAPI()

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    await websocket.send_text("Welcome to WebSocket!")
    data = await websocket.receive_text()
    await websocket.send_text(f"You said: {data}")
```

Connecting to the WebSocket

> Use JavaScript in the frontend to connect to the WebSocket server:



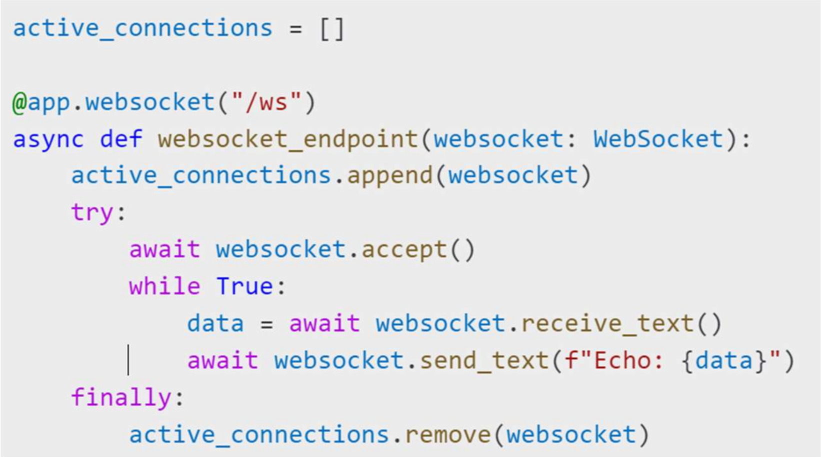
```javascript
const socket = new WebSocket("ws://localhost:8000/ws");

socket.onopen = () => console.log("Connected");
socket.onmessage = (event) => console.log("Message: ", event.data);
socket.onclose = () => console.log("Disconnected");

socket.send("Hello, Server!");
```

Handling Multiple Connections

> Maintain a list of active WebSocket connections:

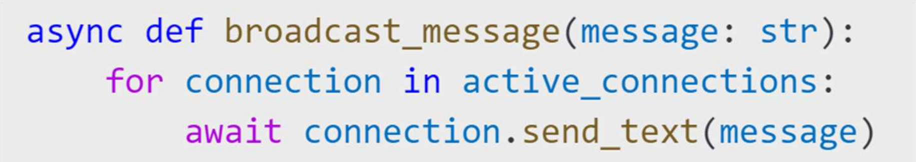


```
active_connections = []

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    active_connections.append(websocket)
    try:
        await websocket.accept()
        while True:
            data = await websocket.receive_text()
            await websocket.send_text(f"Echo: {data}")
    finally:
        active_connections.remove(websocket)
```

Broadcasting Messages

> Send messages to all connected clients:



```
async def broadcast_message(message: str):
    for connection in active_connections:
        await connection.send_text(message)
```

Debugging and Testing WebSockets

> Use browser dev tools to inspect WebSocket frames.

> Tools like wscat for testing:

```
npx wscat -c ws://localhost:8000/ws
```