REST API

Kay Ashaolu - Instructor
Aishwarya Sriram - TA

Chapter 2: A Full Python Refresher

Python Decorators Demystified

- Learn how decorators modify function behavior
- Secure functions without code duplication
- Prepare for integrating these ideas into REST API security

What Are Python Decorators?

- **Definition:** Functions that wrap other functions to extend or modify behavior.
- **Key Benefits:**
  - Eliminate repetitive code (e.g., security checks)
  - Enhance readability and maintainability
- **Basic Concept:** A decorator takes a function as input and returns a modified function.

A Basic Decorator Example

- **Goal:** Secure a function so only admins can call it.
- **Example Code:**

```
def make_secure(func):
    def secure_function(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)
        else:
            return f"No admin permissions for {user['username']}"
    return secure_function

@make_secure
def get_admin_password():
    return "1234"
```

- **Outcome:** `get_admin_password()` now performs an access check.

Enhancing Decorators with "@" Syntax

- **"At" Syntax:** Simplifies applying decorators.

```
@make_secure
def get_admin_password():
    return "1234"
```

- **Preserve Metadata:** Use `functools.wraps` to retain the original function's name and docstring.

```
import functools
def make_secure(func):
    @functools.wraps(func)
    def secure_function(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)
        else:
            return f"No admin permissions for {user['username']}"
    return secure_function
```

Chapter 3: Your First REST API

Building a RESTful Web Service with Flask

- Learn to create endpoints and handle JSON data.
- Develop a simple in-memory data store.
- Understand HTTP methods, status codes, and testing.

Setting Up Your Development Environment

- **Create a Virtual Environment:**

```
python3.10 -m venv .venv
```

- **Activate the Environment:** (Editor-specific configuration)
- **Install Flask:**

```
pip install flask
```

- **Tip:** Use your favorite IDE configured with the correct interpreter.

Initializing a Flask Application

- **File:** `app.py`
- **Basic Boilerplate:**

```
1  from flask import Flask, request
2  app = Flask(__name__)
```

- **Purpose:** Set up the app and prepare to define endpoints.

Running Your Flask App

- **Command:** `flask run`
- **What Happens:**
  - Flask looks for `app.py` and an app variable named `app`.
  - The server runs at `http://127.0.0.1:5000`
- **Note:** Restart the server after code changes.

Creating an In-Memory Data Store

- **Concept:** Use Python lists/dictionaries to store data temporarily.
- **Example Data Structure:**

```
stores = [
    {
        "name": "My Store",
        "items": [
            {"name": "chair", "price": 15.99}
        ]
    }
]
```

- **Usage:** Simulate a database for prototyping your API.

Defining the GET /store Endpoint

- **Objective:** Return all store data as JSON.
- **Endpoint Code:**

```
@app.get("/store")
def get_stores():
    return {"stores": stores}
```

- **Flask Magic:** Automatically converts Python dictionaries to JSON.

What is JSON?

- **Definition:** JavaScript Object Notation, a lightweight data interchange format.
- **Structure:**
  - Key-value pairs (similar to Python dictionaries)
  - Supports arrays, nested objects, and various data types
- **Importance:** Standard format for API communication.

JSON vs. Python Dictionaries

- **JSON:**
  - A string formatted in a specific way.
  - Uses lowercase `true`/`false` and double quotes.
- **Python Dictionary:**
  - An in-memory data structure.
  - Uses Python's `True`/`False` and single/double quotes.
- **Conversion:** Flask automatically serializes dictionaries to JSON.

Testing Your REST API: Tools and Techniques

- **Why Test?**
  - Ensure endpoints work as expected.
  - Catch errors early during development.
- **Tools:**
  - Insomnia
  - Postman
  - Browser (for simple GET requests)
- **Tip:** Start with manual exploratory testing.

Using Insomnia to Test GET Requests

- **Setup:**
  - Create a new request in Insomnia.
  - Set the HTTP method to GET.
  - URL: `http://127.0.0.1:5000/store`
- **What to Expect:**
  - JSON response containing your store data.
  - Ability to view and debug response payloads.

Creating a Store: POST /store Endpoint

- **Objective:** Allow clients to add new stores.
- **Client Action:** Send a JSON payload with the store name.
- **Example JSON:**

```
{
  "name": "My New Store"
}
```

POST /store Endpoint Implementation

- **Endpoint Code:**

```python
@app.post("/store")
def create_store():
    request_data = request.get_json()
    new_store = {
        "name": request_data["name"],
        "items": []
    }
    stores.append(new_store)
    return new_store, 201
```

- **Explanation:**
  - Extract JSON from the request.
  - Append the new store to the `stores` list.
  - Return the new store with HTTP status 201 (Created).

Creating Items in a Store: Dynamic Routes

- **Goal:** Add items to a specific store.
- **Dynamic URL:** Capture the store name from the URL.
- **Endpoint Pattern:** `POST /store/<name>/item`
- **Client Sends:** JSON payload with item details (name and price).

POST /store/<name>/item

- **Implementation:**

```python
@app.post("/store/<string:name>/item")
def create_item(name):
    request_data = request.get_json()
    for store in stores:
        if store["name"] == name:
            new_item = {
                "name": request_data["name"],
                "price": request_data["price"]
            }
            store["items"].append(new_item)
            return new_item, 201
    return {"message": "Store not found"}, 404
```

- **Key Points:**
  - Iterate over stores to find a matching name.
  - Return 404 if the store does not exist.

Retrieving a Specific Store: GET /store/<name>

- **Purpose:** Return the complete store data for a given store name.
- **Endpoint Code:**

```
@app.get("/store/\<string:name>")
def get_store(name):
    for store in stores:
        if store["name"] == name:
            return store
    return {"message": "Store not found"}, 404
```

- **Usage:** Clients can retrieve detailed information about a specific store.

Retrieving Store Items: GET /store/<name>/item

- **Objective:** Provide a list of items for a specific store.
- **Endpoint Code:**

```python
@app.get("/store/<string:name>/item")
def get_store_items(name):
    for store in stores:
        if store["name"] == name:
            return {"items": store["items"]}
    return {"message": "Store not found"}, 404
```

- **Design Note:** Returning an object allows future expansion (e.g., adding metadata).

Design Considerations & Error Handling

- **Returning JSON Objects vs. Lists:**
  - Objects offer flexibility (e.g., adding messages or metadata).
  - Consistency in API responses aids client development.
- **Error Handling:**
  - Use proper HTTP status codes (e.g., 404 for not found).
  - Provide informative messages to guide API users.
- **Scalability:** Prepare for future changes like integrating a database.

Summary & Next Steps

- **Key Takeaways:**
  - **Chapter 2:** Mastered Python decorators to secure and extend function behavior.
  - **Chapter 3:** Built a basic REST API with Flask using GET and POST endpoints.
  - JSON is essential for client-server communication.
  - Testing with tools like Insomnia ensures reliability.