REST API Part II

Kay Ashaolu - Instructor

Aishwarya Sriram - TA

Aside: Errors in Python

Understanding and Handling Errors

The Problem with Print Statements

- Using print for error messages mixes math logic with program context
- Example: In a grading program, printing "divisor cannot be zero" confuses users
- **Solution:** Use errors to signal exceptional conditions

Raising and Handling Errors

```python
def divide(dividend, divisor):
    if divisor == 0:
        raise ZeroDivisionError("divisor cannot be zero")
    return dividend / divisor

try:
    result = divide(sum(grades), len(grades))
except ZeroDivisionError as e:
    print("There are no grades yet in your list.")
```

- **Raise:** Signal error conditions
- **try/except:** Catch and handle exceptions

Chapter 5: Flask-Smorest for Efficient API Development

- Transition from simple API to a well-structured REST API
- Enhancing data models, error handling, and documentation
- Introducing Flask-Smorest, Blueprints, and Marshmallow

Moving from Names to Unique IDs

- **Old Approach:** Use store names as identifiers
- **New Approach:** Use unique IDs (UUIDs or auto-increment numbers)
- **Benefits:**
  - Direct access using dictionaries
  - Simplified and efficient data retrieval

Data Model Transformation

Before:

```
stores = [store1, store2, ...]
```

After:

```
stores = {
    store_id1: { ... },
    store_id2: { ... }
}
```

- No more iterating through lists; access data by unique keys

Creating the Database Module

File: db.py

```
stores = {}
items = {}
```

- Centralizes data storage
- Easily imported into API endpoints

Updating API Endpoints for Unique IDs

```python
@app.get("/store/<string:store_id>")
def get_store(store_id):
    try:
        return stores[store_id]
    except KeyError:
        abort(404, message="Store not found.")
```

- Replace name-based endpoints with ID-based ones
- Simplifies code and error handling

Improved Error Handling with Flask-Smorest

- Use the `abort` function from Flask-Smorest
- Automatically includes error info in the API documentation

```
1 from flask_smorest import abort
2 abort(404, message="Store not found.")
```

Creating Robust Endpoints: CRUD Operations

- **CRUD:** Create, Read, Update, Delete
- Organize endpoints logically for items and stores
- Example for deleting an item:

```python
@app.delete("/item/<string:item_id>")
def delete_item(item_id):
    try:
        del items[item_id]
        return {"message": "Item deleted."}
    except KeyError:
        abort(404, message="Item not found.")
```

Organizing Your API Project

- Group endpoints into folders (e.g., `/items`, `/stores`)
- Keeps the codebase maintainable
- Align endpoint organization with API clients (e.g., Insomnia)

Running the API in Docker

- **Why Docker?**
  - Mimics production environment
  - Avoids “works on my machine” issues
- Update Dockerfile to install dependencies from `requirements.txt`

Enabling Hot Reloading with Docker Volumes

```
1 docker run -dp 5005:5000 -v "$PWD":/app flask-smorest-api
```

- Maps local directory to container's `/app`
- Automatically updates container when code changes

Introducing Blueprints and MethodViews

- **Blueprints:** Modularize API endpoints
- **MethodViews:** Map HTTP methods to class methods

```python
from flask.views import MethodView
from flask_smorest import Blueprint

blp = Blueprint("stores", __name__, description="Operations on stores")

@blp.route("/store/<string:store_id>")
class Store(MethodView):
    def get(self, store_id):
        # Retrieve store logic
        pass

    def delete(self, store_id):
        # Delete store logic
        pass
```

Registering Blueprints in Your Flask App

- Import Blueprints and register them with Flask-Smorest's API

```python
from flask_smorest import Api
from resources.item import blp as ItemBlueprint
from resources.store import blp as StoreBlueprint

api = Api(app)
api.register_blueprint(ItemBlueprint)
api.register_blueprint(StoreBlueprint)
```

- Integrates all endpoints and documentation

Introduction to Marshmallow Schemas

- **Purpose:** Validate incoming data and serialize outgoing responses
- **Example Item Schema:**

```python
from marshmallow import Schema, fields

class ItemSchema(Schema):
    id = fields.Str(dump_only=True)
    name = fields.Str(required=True)
    price = fields.Float(required=True)
    store_id = fields.Str(required=True)
```

Validating API Requests with Marshmallow

- Use `@blp.arguments` to validate incoming JSON data

```
@blp.arguments(ItemSchema)
def post(self, item_data):
    # Process validated data
    pass
```

- Eliminates manual if-checks for required fields

Decorating Responses with Marshmallow

- Use `@blp.response` to format outgoing responses

```
1 @blp.response(200, ItemSchema)
2 def get(self, item_id):
3     # Return item serialized by ItemSchema
4     pass
```

- Enhances documentation and consistency of responses

Handling Multiple Items with Marshmallow

- When returning lists, specify `many=True`

```
1 @blp.response(200, ItemSchema(many=True))
2 def get(self):
3     return list(items.values())
```

- Automatically serializes a list of items into JSON

Enhanced API Documentation

- Flask-Smorest integrates with Swagger UI
- Automatically documents endpoints, schemas, and responses
- Example: Access API docs at `/swagger-ui`

Benefits of Using Flask-Smorest & Marshmallow

- **Structured Code:** Modular, maintainable endpoints
- **Validation:** Clear schema-based validation
- **Documentation:** Up-to-date API docs via Swagger UI
- **Error Handling:** Consistent error messages and status codes

Best Practices Recap

- Transition to unique identifiers for data models
- Organize endpoints with Blueprints and MethodViews
- Validate and serialize data with Marshmallow
- Use Docker for consistent development environments

Recap: Chapter 5 Highlights

- **Data Models:** Efficient lookup using dictionaries
- **Error Handling:** Centralized and documented via Flask-Smorest
- **Blueprints & MethodViews:** Clean, modular API structure
- **Marshmallow:** Robust data validation and serialization

Full Example

Bringing together serialization and deserialization

- Use **Marshmallow Schemas** to validate incoming data (deserialization)
- Use schemas to format outgoing responses (serialization)
- Organize endpoints with **Flask-Smorest Blueprints** and **MethodViews**

Item Resource Blueprint with Marshmallow

```python
# resources/item.py
from flask.views import MethodView
from flask_smorest import Blueprint, abort
import uuid
from schemas import ItemSchema, ItemUpdateSchema

# In-memory storage for demonstration
items = {}

blp = Blueprint("items", __name__, description="Operations on items")

@blp.route("/item/<string:item_id>")
class Item(MethodView):
    @blp.response(200, ItemSchema)
    def get(self, item_id):
        try:
            return items[item_id]
        except KeyError:
            abort(404, message="Item not found.")
```

```python

    @blp.arguments(ItemUpdateSchema)
    @blp.response(200, ItemSchema)
    def put(self, item_data, item_id):
        try:
            item = items[item_id]
        except KeyError:
            abort(404, message="Item not found.")
        item.update(item_data)
        return item

    def delete(self, item_id):
        try:
            del items[item_id]
            return {"message": "Item deleted."}
        except KeyError:
            abort(404, message="Item not found.")
```

```python

@blp.route("/item")
class ItemList(MethodView):
    @blp.response(200, ItemSchema(many=True))
    def get(self):
        return list(items.values())

    @blp.arguments(ItemSchema)
    @blp.response(201, ItemSchema)
    def post(self, item_data):
        item_id = uuid.uuid4().hex
        item = {**item_data, "id": item_id}
        items[item_id] = item
        return item
```

- **@blp.arguments:** Validates and deserializes incoming JSON using a schema
- **@blp.response:** Serializes outgoing data using a schema

Example Summary

- **Deserialization:**
  - Incoming JSON is validated by `ItemSchema` or `ItemUpdateSchema`
  - Ensures required fields and types before processing
- **Serialization:**
  - Outgoing Python objects are formatted as JSON using the same schemas
  - Guarantees consistent API responses and up-to-date documentation
- **Blueprints & MethodViews:**
  - Modular structure for API endpoints
  - Clean separation of HTTP methods (GET, POST, PUT, DELETE) within a class

Questions?