Learning APIs and SOAP with Python



Published by Randy Fadler

July 2025

**📘 Executive Summary**

***Building and Using APIs & SOAP Messages: A Practical Guide*** is a hands-on, example-driven booklet designed to help software engineers, technical educators, and integration professionals master the art of API development and interoperability. Blending clarity, warmth, and depth, this guide demystifies both modern RESTful services and enterprise-grade SOAP messaging through real-world scenarios, fully functional round-trip code examples, and step-by-step integration workflows.

Spanning six detailed chapters plus an extended appendix, the booklet guides readers through:

* Building REST APIs with frameworks like Flask and ASP.NET Core
* Implementing full CRUD operations and request-response flows
* Securing endpoints using API keys, JWT, OAuth2, and WS-Security headers
* Documenting services using Swagger/OpenAPI and Postman
* Testing and monitoring API health using manual and automated tools
* Integrating REST and SOAP endpoints within front-end apps and orchestration services

Each concept is scaffolded to reinforce learning, anticipating learner confusion and offering clear, contextual code to build confidence and skill. The running example—a modular Book Info API with an extended Customer Feedback Analyzer—showcases cross-platform and cross-protocol patterns, from browser-based clients to SOAP proxying.

Whether onboarding new developers, retooling legacy systems, or teaching API fundamentals in the classroom, this guide offers practical depth and pedagogical care that makes the invisible mechanics of system communication intuitive, accessible, and highly relevant.

Table of Contents

[🧭 Introduction: Why APIs—and SOAP—Still Matter 7](#_Toc203984881)

[🔍 Chapter 1: What Is an API? 8](#_Toc203984882)

[🎯 Chapter Objectives 8](#_Toc203984883)

[🌐 1.1 Defining APIs with Real-Life Analogies 8](#_Toc203984884)

[💻 1.2 Setting Up the Development Environment 8](#_Toc203984885)

[🧱 1.3 Building Your First RESTful API (Flask) 9](#_Toc203984886)

[📨 1.4 Creating the Client Program (requests) 10](#_Toc203984887)

[📚 1.6 Expanding the API 11](#_Toc203984888)

[📖 1.7 Summary and What’s Next 12](#_Toc203984889)

[🌐 Chapter 2: RESTful APIs in Practice 13](#_Toc203984890)

[🎯 Chapter Objectives 13](#_Toc203984891)

[📖 2.1 Understanding RESTful Design 13](#_Toc203984892)

[🧱 2.2 Refactoring the API with Full CRUD Support 13](#_Toc203984893)

[💥 2.4 HTTP Status Codes and Error Management 16](#_Toc203984894)

[🧭 2.5 Debugging the REST Stack 16](#_Toc203984895)

[📚 2.6 Summary 16](#_Toc203984896)

[🧼 Chapter 3: Designing, Documenting, and Testing REST APIs 17](#_Toc203984897)

[🎯 Chapter Objectives 17](#_Toc203984898)

[📘 3.1 Why API Documentation Is Not Optional 17](#_Toc203984899)

[🗂 3.2 Introducing OpenAPI (Swagger) 17](#_Toc203984900)

[🧪 3.4 Testing with Postman 19](#_Toc203984901)

[📐 3.5 Validating Inputs with Marshmallow 20](#_Toc203984902)

[pip install marshmallow 20](#_Toc203984903)

[🔁 3.6 Automated Testing 20](#_Toc203984904)

[🧰 3.7 Best Practices for Real-World APIs 21](#_Toc203984905)

[🧠 Summary and What’s Next 21](#_Toc203984906)

[🔐 Chapter 4: Securing APIs—Authentication and Authorization 22](#_Toc203984907)

[🎯 Chapter Objectives 22](#_Toc203984908)

[🔍 4.1 Auth vs AuthZ—What’s the Difference? 22](#_Toc203984909)

[🔑 4.2 Securing REST APIs with API Keys 22](#_Toc203984910)

[🛂 4.3 JWT: JSON Web Tokens for Stateless Auth 23](#_Toc203984911)

[🔁 4.4 OAuth2: Delegated Authorization 24](#_Toc203984912)

[🧼 4.5 Securing SOAP with WS-Security 25](#_Toc203984913)

[🔐 C# Example: 25](#_Toc203984914)

[🛡 4.6 Best Practices for Securing APIs 25](#_Toc203984915)

[🧠 Summary and Transition 26](#_Toc203984916)

[🎯 Chapter Objectives 27](#_Toc203984917)

[🧬 5.1 What Does “Healthy” Mean for an API? 27](#_Toc203984918)

[⚕️ 5.2 Adding a Healthcheck Endpoint 27](#_Toc203984919)

[🔧 5.3 Manual Testing with Postman and curl 28](#_Toc203984920)

[🧪 5.4 Automated Testing with pytest 28](#_Toc203984921)

[🧱 5.5 Introducing Logging and Observability 29](#_Toc203984922)

[📡 5.6 External Monitoring Tools 30](#_Toc203984923)

[🧪 5.7 Rate Limits, Timeouts, and Error Recovery 30](#_Toc203984924)

[📖 5.8 Writing Tests for Edge Cases 31](#_Toc203984925)

[🧠 Summary and Transition 31](#_Toc203984926)

[🧩 Chapter 6: Integrating APIs Into Real Applications 32](#_Toc203984927)

[📘 6.1 Integration Isn’t Just Connection—It’s Coordination 32](#_Toc203984928)

[🛠️ 6.2 Building a Front-End Interface 32](#_Toc203984929)

[🛠️ 6.3 Orchestrating Multiple REST Calls 33](#_Toc203984930)

[🌐 6.4 Integrating a SOAP Service Into REST Workflow 33](#_Toc203984931)

[📦 6.5 Versioning and Maintainability 34](#_Toc203984932)

[🧪 6.6 Documentation and Integration Help 34](#_Toc203984933)

[🎓 6.7 Bringing It All Together: Customer Feedback Analyzer 35](#_Toc203984934)

[📚 Appendix: Cheat Sheets, Glossary, and Further Reading 36](#_Toc203984935)

[🗂️ REST & SOAP Command Cheat Sheets 36](#_Toc203984936)

[🧼 Common SOAP Elements 36](#_Toc203984937)

[🗣️ Glossary of API Terms 37](#_Toc203984938)

[💾 Full Code Suite: Building & Using APIs and SOAP Messages 39](#_Toc203984939)

[📁 Project Structure Overview 39](#_Toc203984940)

[📘 Chapter 1: Basic REST API and Client 39](#_Toc203984941)

[📘 Chapter 2: CRUD API with Client 40](#_Toc203984942)

[📘 Chapter 3: Swagger + Testing 42](#_Toc203984943)

[📘 Chapter 4: Security with API Key + JWT 43](#_Toc203984944)

[📘 Chapter 6: Orchestration + SOAP Proxy + UI 44](#_Toc203984945)

**📘 Booklet Title: *Building and Using APIs & SOAP Messages: A Practical Guide***

**🧭 Introduction**

* **Purpose of the Booklet** Explain the relevance of APIs in modern software ecosystems and why SOAP still matters—especially in healthcare, finance, and enterprise systems.
* **Who It's For** Target developers, educators, and professionals needing hands-on API guidance.
* **How to Use This Booklet** Introduce your running example methodology and how readers can build incrementally.

**🔍 Chapter 1: What Is an API?**

* **API Fundamentals** Define APIs with analogies (e.g., waiter-restaurant metaphor) and highlight different types: REST, SOAP, GraphQL.
* **Common Use Cases** Anchor in real-world examples: weather apps, payment processing, interoperability between healthcare systems.
* **Hands-On Exercise** Create a simple local API in Flask or ASP.NET and call it from Python or C#.

💬 *Comment:* This sets the tone for learners. Consider layering theory with immediate code walkthroughs.

**🌐 Chapter 2: RESTful APIs in Practice**

* **HTTP Methods & Status Codes** Explore GET, POST, PUT, DELETE, and common responses like 200, 404, 500.
* **Building a REST API** Use Python Flask or C# Web API. Include routing, JSON serialization, error handling.
* **Calling External REST APIs** Include curl, Python requests, and C# HttpClient examples.
* **Debugging & Troubleshooting** Guide through common pitfalls like CORS issues, malformed payloads, and authentication errors.

💬 *Comment:* Emphasize scaffolding here—each endpoint builds toward a full app.

**🧼 Chapter 3: Working with SOAP Messages**

* **What Is SOAP and Why It's Still Around** Contextualize SOAP’s role in enterprise and healthcare (e.g., HL7).
* **Structure of a SOAP Envelope** Break down headers, body, and namespaces visually.
* **Creating SOAP Requests** Use Postman and then code examples in Python (zeep) and C# (SoapClient or HttpWebRequest).
* **Parsing SOAP Responses** Walk through interpreting XML payloads and error handling.

💬 *Comment:* Use contrast with REST to highlight SOAP’s verbosity and formality.

**🔐 Chapter 4: Authentication and Security**

* **API Keys, OAuth2, and JWTs** Explain flows and when each is appropriate.
* **SOAP WS-Security** Cover headers, signatures, and how enterprise systems secure legacy messages.
* **Practical Setup** Provide reusable boilerplate with security baked in for both REST and SOAP.

💬 *Comment:* Learners often struggle here—clarify security pitfalls early.

**🔧 Chapter 5: Testing and Monitoring APIs**

* **Tools: Postman, Swagger, SoapUI, Fiddler** Showcase key features with screenshots or annotated walkthroughs.
* **Automated Tests with PyTest / NUnit** Include code examples for regression testing endpoints.
* **Monitoring & Logging Tips** Guide through logging requests and setting up error alerts.

💬 *Comment:* Encourage best practices so learners go beyond “just make it work.”

**🧩 Chapter 6: Integrating APIs in Real Projects**

* **Customer Feedback Analyzer** (Running Example) Demonstrate REST and SOAP endpoints working together in one app.
* **Front-End Integration** Example with React or simple HTML form calling APIs.
* **Versioning & Documentation Tips** Introduce OpenAPI/Swagger and strategies for maintainability.

💬 *Comment:* This chapter exemplifies your "bridge-the-gap" philosophy between theory and practice.

**📚 Appendix**

* **Glossary of Terms**
* **Cheat Sheets for REST and SOAP Commands**
* **Recommended Resources** Link to HL7 spec, Zeep docs, Microsoft Learn, Postman tutorials.

# 🧭 Introduction: Why APIs—and SOAP—Still Matter

In today’s software landscape, APIs (Application Programming Interfaces) are not just a convenience—they’re the connective tissue of nearly every digital experience. Whether you’re ordering a coffee via a mobile app, syncing your fitness tracker to the cloud, or retrieving lab results from an electronic health record (EHR), an API is quietly doing the heavy lifting behind the scenes. RESTful APIs dominate the modern tech stack with their lightweight design, JSON payloads, and ease of integration. But SOAP (Simple Object Access Protocol), despite its age and verbosity, remains crucial in enterprise environments—particularly those driven by regulatory standards, strict validation requirements, and long-standing interoperability protocols like HL7 v2 or WSDL contracts.

This booklet is designed for developers and educators who want to build a well-rounded understanding of APIs—not just trendy REST services, but also the more formal SOAP mechanisms that power critical systems in healthcare, finance, logistics, and government. Too many tutorials gloss over the real-world messiness—authentication headaches, confusing payload schemas, legacy constraints, or integration across multi-platform front ends. Here, we face those head-on.

We’ll walk through building REST APIs using frameworks like Python’s Flask and C#’s ASP.NET Core, complete with routing, model binding, and error handling. You’ll learn how to consume these services using tools like curl, Postman, or HttpClient. Then we’ll pivot to SOAP: how to craft and validate XML envelopes, manage namespaces, and transmit secure payloads with WS-Security headers. We’ll use examples like pulling patient demographics from a SOAP-enabled HL7 broker or submitting claims data to a government portal.

Everything is scaffolded—each concept builds on the last. You’ll explore how endpoints integrate into full-stack apps, how to debug malformed requests, and how to keep your APIs secure and well-documented. Expect practical exercises, annotated code, and worked-through running examples like a Customer Feedback Analyzer that brings REST and SOAP together under one roof.

Whether you're teaching students, onboarding junior devs, or switching stacks yourself, this guide aims to bridge theory and practice—always with clarity, humor, and real-world payoff.

## 🔍 Chapter 1: What Is an API?

## 🎯 Chapter Objectives

By the end of this chapter, learners will:

* Grasp the conceptual role of APIs in system interoperability.
* Understand HTTP protocol basics including GET requests.
* Build and run a live RESTful API using Python’s Flask framework.
* Create a client-side script to interact with the API.
* Diagnose and resolve common beginner errors in API consumption.

## 🌐 1.1 Defining APIs with Real-Life Analogies

An API (Application Programming Interface) is a digital doorway into the functionality or data of an application. Just as a restaurant patron communicates with the kitchen via a waiter, software communicates via APIs.

**🤔 Real-world analogies:**

* **Waiter Analogy**: The customer (client) places an order (request), the waiter (API) relays it to the kitchen (server), and the meal (response) is served.
* **Power Outlet**: Devices plug into an outlet without worrying how electricity is generated—similar to how clients use APIs without knowing internal logic.
* **ATM Machine**: A structured interface gives access to bank data with known inputs (card + PIN) and expected outputs (balance, withdrawal).

💬 Learner Tip: Always think about what the client knows vs. what the server is doing behind the scenes. APIs abstract complexity.

## 💻 1.2 Setting Up the Development Environment

Before diving into code, learners need a working Python environment with Flask installed.

**✅ Requirements:**

* Python 3.8+
* pip
* Flask

**🔧 Setup Commands:**

bash

# Install Flask

pip install flask

💡 Optional: Recommend using a virtual environment:

bash

python -m venv api\_env

source api\_env/bin/activate # Mac/Linux

api\_env\Scripts\activate # Windows

## 🧱 1.3 Building Your First RESTful API (Flask)

Let’s create a REST API that serves book information based on an ID.

**📁 Project Structure:**

api\_project/

│

├── book\_api.py # Flask-based server-side API

├── book\_client.py # Client-side request handler

└── README.md # Documentation and usage tips

**📦 API Code Explained**

python

# book\_api.py

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

# Sample data: pretend this is our tiny database

BOOKS = {

1: {"title": "The Pragmatic Programmer", "author": "Andrew Hunt", "year": 1999},

2: {"title": "Clean Code", "author": "Robert C. Martin", "year": 2008},

3: {"title": "Fluent Python", "author": "Luciano Ramalho", "year": 2015}

}

@app.route('/book/<int:book\_id>', methods=['GET'])

def get\_book(book\_id):

book = BOOKS.get(book\_id)

if book:

return jsonify(book) # Converts Python dict to JSON

return jsonify({"error": "Book not found"}), 404

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**🚀 Running the API**

bash

python book\_api.py

You should see:

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

Try visiting in browser:

http://127.0.0.1:5000/book/1

You’ll get:

json

{"title":"The Pragmatic Programmer","author":"Andrew Hunt","year":1999}

## 📨 1.4 Creating the Client Program (requests)

The client needs to send a request and interpret the response.

python

# book\_client.py

import requests

def fetch\_book(book\_id):

url = f'http://127.0.0.1:5000/book/{book\_id}'

try:

response = requests.get(url)

response.raise\_for\_status()

book = response.json()

print(f"\n Title: {book['title']}")

print(f" Author: {book['author']}")

print(f" Year: {book['year']}")

except requests.exceptions.HTTPError as err:

print(f"❌ API Error: {err}")

except Exception as e:

print(f"❗ General Error: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

fetch\_book(2)

🔄 Learner Insight: This client demonstrates both consumption and basic error handling. You can expand this to loop through multiple IDs or handle different endpoints later.

🔁 1.5 Round-Trip Interaction Explained

* **Client** initiates an HTTP GET to /book/<id>.
* **API** responds with a structured JSON object.
* **Client** parses the JSON and renders the result to console.

**🔍 What to Debug If It Fails**

| **Problem** | **Likely Cause** | **Fix** |
| --- | --- | --- |
| ConnectionRefusedError | API not running | Start Flask server |
| 404 Error | Wrong book ID or malformed route | Check URL and ID |
| JSON decode error | Non-JSON response or empty | Add debug logging |

## 📚 1.6 Expanding the API

Challenge learners to add:

* 🔸 A **POST endpoint** to add a new book.
* 🔸 **Query parameters** for searching by author.
* 🔸 Structured logging using logging module.
* 🔸 A basic README.md for documentation.

## 📖 1.7 Summary and What’s Next

This chapter laid the foundation with:

* A functioning API exposed via Flask.
* A companion client program demonstrating API consumption.
* Error handling, data serialization, and endpoint routing.

Next up: we’ll enrich the API with multiple endpoints, add POST requests, handle authentication, and introduce Swagger for documentation.

# 🌐 Chapter 2: RESTful APIs in Practice

## 🎯 Chapter Objectives

By the end of this chapter, readers will:

* Understand how REST maps to CRUD operations via HTTP methods.
* Implement endpoints using GET, POST, PUT, DELETE in Flask.
* Handle errors gracefully and return meaningful response codes.
* Modify the client program to interact with all endpoints.
* Anticipate and debug common REST pitfalls in real-world development.

## 📖 2.1 Understanding RESTful Design

REST stands for *Representational State Transfer*. It’s an architectural style where systems communicate over HTTP using **resources** (like “books”) and **verbs** (GET, POST, etc.). Each URL maps to a resource; each HTTP method defines what to do with it.

**🔧 Mapping CRUD to REST:**

| **CRUD Operation** | **HTTP Method** | **Endpoint Example** | **Action** |
| --- | --- | --- | --- |
| Create | POST | /book | Add a new book |
| Read | GET | /book/<id> | Retrieve book by ID |
| Update | PUT | /book/<id> | Modify existing book |
| Delete | DELETE | /book/<id> | Remove book from collection |

Think of endpoints as nouns (resources) and methods as verbs (actions).

## 🧱 2.2 Refactoring the API with Full CRUD Support

Let’s evolve book\_api.py into a robust RESTful service.

python

# file: book\_api.py

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

BOOKS = {

1: {"title": "The Pragmatic Programmer", "author": "Andrew Hunt", "year": 1999},

2: {"title": "Clean Code", "author": "Robert C. Martin", "year": 2008}

}

# GET: Retrieve book by ID

@app.route('/book/<int:book\_id>', methods=['GET'])

def get\_book(book\_id):

book = BOOKS.get(book\_id)

if book:

return jsonify(book)

return jsonify({"error": "Book not found"}), 404

# POST: Add a new book

@app.route('/book', methods=['POST'])

def add\_book():

data = request.json

if not data or not all(k in data for k in ("title", "author", "year")):

return jsonify({"error": "Missing required fields"}), 400

new\_id = max(BOOKS) + 1

BOOKS[new\_id] = {

"title": data["title"],

"author": data["author"],

"year": data["year"]

}

return jsonify({"message": "Book added", "book\_id": new\_id}), 201

# PUT: Update existing book

@app.route('/book/<int:book\_id>', methods=['PUT'])

def update\_book(book\_id):

if book\_id not in BOOKS:

return jsonify({"error": "Book not found"}), 404

data = request.json

BOOKS[book\_id].update(data)

return jsonify({"message": "Book updated", "book": BOOKS[book\_id]})

# DELETE: Remove a book

@app.route('/book/<int:book\_id>', methods=['DELETE'])

def delete\_book(book\_id):

if book\_id not in BOOKS:

return jsonify({"error": "Book not found"}), 404

del BOOKS[book\_id]

return jsonify({"message": f"Book {book\_id} deleted"})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**🧪 2.3 Enhancing the Client for CRUD Interaction**

Let’s now write a **modular client** that performs all four operations.

python

# file: book\_client.py

import requests

BASE\_URL = 'http://127.0.0.1:5000/book'

def create\_book(title, author, year):

payload = {"title": title, "author": author, "year": year}

res = requests.post(BASE\_URL, json=payload)

print(res.json())

def get\_book(book\_id):

res = requests.get(f"{BASE\_URL}/{book\_id}")

print(res.json())

def update\_book(book\_id, updates):

res = requests.put(f"{BASE\_URL}/{book\_id}", json=updates)

print(res.json())

def delete\_book(book\_id):

res = requests.delete(f"{BASE\_URL}/{book\_id}")

print(res.json())

if \_\_name\_\_ == "\_\_main\_\_":

print("\n📘 Creating a new book:")

create\_book("Fluent Python", "Luciano Ramalho", 2015)

print("\n📘 Retrieving book ID 3:")

get\_book(3)

print("\n📘 Updating book ID 3:")

update\_book(3, {"year": 2016})

print("\n📘 Deleting book ID 3:")

delete\_book(3)

## 💥 2.4 HTTP Status Codes and Error Management

APIs should communicate outcomes clearly with status codes.

| **Code** | **Meaning** | **When to Use** |
| --- | --- | --- |
| 200 | OK | Successful GET/PUT/DELETE |
| 201 | Created | Successful POST |
| 400 | Bad Request | Missing fields, malformed |
| 404 | Not Found | Resource doesn’t exist |
| 500 | Server Error | Unexpected exceptions |

🧠 Note for Learners: Always validate inputs on both client and server side to avoid 400 errors.

## 🧭 2.5 Debugging the REST Stack

**🧪 Tools Worth Learning:**

* **Postman**: Test endpoints with GUI.
* **curl**: Command-line testing.
* **Swagger UI**: Visual API explorer (covered in Chapter 3).
* **Logging**: Add logging module in Flask for runtime insights.

✅ Challenge: Extend the API to support searching books by author using query parameters: /book?author=Martin.

## 📚 2.6 Summary

This chapter grounded learners in RESTful API development with real CRUD functionality. By running both API and client locally, readers experience the entire lifecycle: request formation, server routing, response payloads, and error interpretation.

# 🧼 Chapter 3: Designing, Documenting, and Testing REST APIs

## 🎯 Chapter Objectives

Learners will:

* Understand why API documentation matters.
* Create Swagger/OpenAPI definitions manually and dynamically using Flask.
* Use Postman for exploration, manual testing, and scripting API requests.
* Validate input using schema libraries (e.g., Marshmallow).
* Test endpoints using automated tools (pytest, unittest).
* Learn best practices for maintainable, well-documented APIs.

## 📘 3.1 Why API Documentation Is Not Optional

In team environments, APIs without documentation are bottlenecks. They increase integration time, miscommunication, and bugs.

**💡 Analogy: API Docs as “Menus”**

Imagine walking into a restaurant with no menu—how do you know what you can order, or what ingredients are available? Swagger and Postman are your API menus.

## 🗂 3.2 Introducing OpenAPI (Swagger)

OpenAPI (formerly Swagger) is a specification for documenting REST APIs. You describe endpoints, payloads, query parameters, status codes, and authentication strategies.

Here’s how a simple endpoint looks in Swagger YAML:

yaml

paths:

/book/{book\_id}:

get:

summary: Get book info by ID

parameters:

- in: path

name: book\_id

required: true

schema:

type: integer

responses:

'200':

description: Successful response

content:

application/json:

example:

title: "Clean Code"

author: "Robert C. Martin"

year: 2008

'404':

description: Book not found

**⚙️ 3.3 Auto-Generating Swagger in Flask**

Install flasgger, a plugin that auto-generates Swagger UI.

bash

pip install flasgger

Update your API:

python

# book\_api\_swagger.py

from flask import Flask, request, jsonify

from flasgger import Swagger

app = Flask(\_\_name\_\_)

swagger = Swagger(app)

BOOKS = { ... }

@app.route('/book/<int:book\_id>', methods=['GET'])

def get\_book(book\_id):

"""

Get Book Info by ID

---

parameters:

- name: book\_id

in: path

type: integer

required: true

responses:

200:

description: Book found

schema:

id: Book

properties:

title:

type: string

author:

type: string

year:

type: integer

404:

description: Book not found

"""

...

🔗 Visit http://127.0.0.1:5000/apidocs for a live Swagger UI interface.

## 🧪 3.4 Testing with Postman

Postman is a GUI tool for sending requests, scripting flows, and validating outputs.

**🧰 Quick Guide:**

* **Set Method**: Choose GET, POST, etc.
* **Set URL**: http://127.0.0.1:5000/book/2
* **Headers**: Add Content-Type: application/json
* **Body**: JSON for POST/PUT requests

**🔄 Example POST Request:**

json

{

"title": "Refactoring",

"author": "Martin Fowler",

"year": 1999

}

Postman also allows:

* **Environment Variables**
* **Test Scripts** (e.g., assert response status)
* **Collections** for bundling API workflows

## 📐 3.5 Validating Inputs with Marshmallow

Sanitize and validate user input to prevent bad payloads or security flaws.

bash

## pip install marshmallow

**🛡 Schema Example:**

python

from marshmallow import Schema, fields, ValidationError

class BookSchema(Schema):

title = fields.Str(required=True)

author = fields.Str(required=True)

year = fields.Int(required=True)

book\_schema = BookSchema()

@app.route('/book', methods=['POST'])

def add\_book():

try:

data = book\_schema.load(request.json)

except ValidationError as err:

return jsonify(err.messages), 400

...

👨‍🏫 Teaching Tip: This helps learners understand defensive programming.

## 🔁 3.6 Automated Testing

Use pytest or unittest to write tests for your endpoints. This boosts confidence, prevents regressions, and facilitates CI/CD pipelines.

**✅ pytest Example:**

python

# test\_book\_api.py

import requests

BASE\_URL = "http://127.0.0.1:5000/book"

def test\_get\_book():

res = requests.get(f"{BASE\_URL}/1")

assert res.status\_code == 200

assert res.json()["title"] == "The Pragmatic Programmer"

def test\_book\_not\_found():

res = requests.get(f"{BASE\_URL}/999")

assert res.status\_code == 404

💬 Encourage learners to write one test per endpoint and edge case.

## 🧰 3.7 Best Practices for Real-World APIs

| **Best Practice** | **Why It Matters** |
| --- | --- |
| Use versioning (/api/v1/book) | Supports future changes |
| Include detailed error messages | Improves debugging |
| Use consistent naming conventions | Boosts clarity |
| Document with Swagger | Helps integrators |
| Validate input | Prevents unexpected behavior |

🛠 Pro Tip: Add a healthcheck endpoint: /ping → {"status": "ok"} for monitoring.

## 🧠 Summary and What’s Next

This chapter taught learners to go beyond basic functionality and build **transparent, testable APIs** with industry-standard documentation and validation. We’ve now got a clean interface, robust testing, and structured input handling.

In Chapter 4, we dive into **authentication and authorization**—from API keys to JWT and OAuth2. We’ll secure our endpoints and explore how different systems (including SOAP) manage trust and identity.

# 🔐 Chapter 4: Securing APIs—Authentication and Authorization

## 🎯 Chapter Objectives

By the end of this chapter, learners will:

* Understand the difference between authentication and authorization.
* Implement simple authentication mechanisms for REST APIs (API key, JWT).
* Set up OAuth2 flows conceptually and practically.
* Learn about WS-Security and XML signatures in SOAP messaging.
* Recognize security flaws and mitigate risks in request handling.

## 🔍 4.1 Auth vs AuthZ—What’s the Difference?

* **Authentication**: *Who are you?* (Validating identity)
* **Authorization**: *Are you allowed to do this?* (Permissions)

🔐 Example:

* A login screen authenticates you.
* Your admin dashboard authorization lets you manage users. Not all users who log in should have the same access.

## 🔑 4.2 Securing REST APIs with API Keys

This method is simple but not bulletproof. You pass an API key in the request headers.

**🔧 Flask Implementation:**

python

# book\_api\_secure.py

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

API\_KEY = "SECRET123"

def require\_api\_key(func):

def wrapper(\*args, \*\*kwargs):

key = request.headers.get('X-API-KEY')

if key != API\_KEY:

return jsonify({"error": "Unauthorized"}), 401

return func(\*args, \*\*kwargs)

wrapper.\_\_name\_\_ = func.\_\_name\_\_

return wrapper

@app.route('/book/<int:book\_id>', methods=['GET'])

@require\_api\_key

def get\_book(book\_id):

...

**🔐 Calling It:**

python

# book\_client\_auth.py

headers = {"X-API-KEY": "SECRET123"}

res = requests.get('http://127.0.0.1:5000/book/2', headers=headers)

print(res.json())

🚨 Pitfall: Exposed API keys can be leaked in browser or logs. Consider this a beginner method.

## 🛂 4.3 JWT: JSON Web Tokens for Stateless Auth

JWTs are digitally signed tokens passed from client to server. They encode user identity and claims.

**📦 Token Structure:**

text

HEADER.PAYLOAD.SIGNATURE

**🔧 Setup (Flask + PyJWT):**

bash

pip install pyjwt

**🔐 Issuing a Token:**

python

import jwt, datetime

def generate\_token(username):

payload = {

"user": username,

"exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1)

}

token = jwt.encode(payload, "SECRET\_KEY", algorithm="HS256")

return token

**📋 Verifying Token:**

python

def require\_token(func):

def wrapper(\*args, \*\*kwargs):

token = request.headers.get('Authorization')

if not token:

return jsonify({"error": "Missing token"}), 401

try:

jwt.decode(token, "SECRET\_KEY", algorithms=["HS256"])

except jwt.ExpiredSignatureError:

return jsonify({"error": "Expired token"}), 403

except jwt.InvalidTokenError:

return jsonify({"error": "Invalid token"}), 403

return func(\*args, \*\*kwargs)

wrapper.\_\_name\_\_ = func.\_\_name\_\_

return wrapper

💡 Learner Tip: JWT payloads are base64 encoded—not encrypted.

## 🔁 4.4 OAuth2: Delegated Authorization

OAuth2 allows third-party apps to access user data without exposing credentials. Think: "Login with Google."

**👣 Flow Summary:**

1. Client requests access
2. User authorizes
3. Authorization server issues token
4. Client uses token to make API calls

🛠 Tools: Auth0, Okta, Azure AD simplify OAuth2 for learners. Setup will vary, so this booklet includes conceptual flowcharts rather than full code.

## 🧼 4.5 Securing SOAP with WS-Security

SOAP isn’t REST—it doesn’t use headers the same way. Instead, it relies on **WS-Security**, an XML-based standard for embedding security directly into the message.

**🔧 Sample SOAP Envelope:**

xml

<soap:Header>

<wsse:Security>

<wsse:UsernameToken>

<wsse:Username>randy</wsse:Username>

<wsse:Password>p@ssw0rd</wsse:Password>

</wsse:UsernameToken>

</wsse:Security>

</soap:Header>

🛡 Pro Tip: For HL7 v3 or government exchanges, use digital certificates, not plain passwords.

**📦 Python with zeep:**

python

from zeep.wsse.username import UsernameToken

client = Client(wsdl\_file, wsse=UsernameToken("randy", "p@ssw0rd"))

## 🔐 C# Example:

csharp

BasicHttpBinding binding = new BasicHttpBinding();

binding.Security.Mode = BasicHttpSecurityMode.TransportWithMessageCredential;

client.ClientCredentials.UserName.UserName = "randy";

client.ClientCredentials.UserName.Password = "p@ssw0rd";

## 🛡 4.6 Best Practices for Securing APIs

| **Practice** | **Description** |
| --- | --- |
| Use HTTPS | Prevents packet sniffing |
| Avoid hardcoded secrets | Use environment variables or vaults |
| Apply rate limiting | Protect against abuse |
| Sanitize inputs | Prevent XSS and injection attacks |
| Log suspicious activity | Detect brute force or token misuse |
| Rotate keys and tokens | Expired credentials limit exposure |

👨‍🏫 Educator Note: Reinforce security early—even in toy projects. It's a mindset.

## 🧠 Summary and Transition

In this chapter, you added meaningful security layers to your API. From basic keys to JWT and WS-Security, you now know how to authenticate users, validate access, and protect data in transit.

Up next in **Chapter 5**, we explore **monitoring and testing APIs**. You’ll use automated tests, set up health checks, and monitor performance with logging frameworks and external tools.

**🧪 Chapter 5: Testing and Monitoring Your API Ecosystem**

## 🎯 Chapter Objectives

Learners will:

* Implement manual and automated testing of API endpoints.
* Understand the role of health checks and observability.
* Use tools like Postman, curl, pytest, and Swagger for diagnostics.
* Add structured logging and traceability to their API.
* Set up basic monitoring alerts and strategies for resilience.

## 🧬 5.1 What Does “Healthy” Mean for an API?

Healthy APIs are:

* **Responsive** (low latency)
* **Predictable** (reliable response codes and payloads)
* **Observable** (transparent logging, errors aren't hidden)
* **Testable** (each endpoint behaves as documented)

🧠 Analogy: Think of APIs as organs in a digital body—they must show signs of life, respond to stimuli, and report when something goes wrong.

## ⚕️ 5.2 Adding a Healthcheck Endpoint

A healthcheck endpoint is a lightweight probe to verify service availability.

python

@app.route('/ping', methods=['GET'])

def ping():

return jsonify({"status": "ok"}), 200

Use it in:

* Kubernetes readiness probes
* Cloud provider health dashboards
* Basic client availability checks

bash

curl <http://localhost:5000/ping>

📍 Expand with diagnostic metadata (e.g., uptime, version number, database availability).

## 🔧 5.3 Manual Testing with Postman and curl

**🧪 Using curl**

bash

curl -X GET http://localhost:5000/book/2

curl -X POST -H "Content-Type: application/json" \

-d '{"title":"Refactoring","author":"Martin Fowler","year":1999}' \

<http://localhost:5000/book>

**🧪 Using Postman**

* Set headers, request body, and expected response.
* Create Collections to save grouped tests.
* Use Tests tab to write assertions:

javascript

pm.test("Status code is 200", function () {

pm.response.to.have.status(200);

});

## 🧪 5.4 Automated Testing with pytest

Automated tests catch regressions early and validate consistency.

**📁 Recommended Structure:**

api\_project/

├── tests/

│ └── test\_book\_api.py

**✅ Sample Tests**

python

import requests

def test\_get\_valid\_book():

res = requests.get("http://localhost:5000/book/1")

assert res.status\_code == 200

assert "title" in res.json()

def test\_get\_invalid\_book():

res = requests.get("http://localhost:5000/book/999")

assert res.status\_code == 404

assert "error" in res.json()

Run with:

bash

pytest tests/

🧪 Tip: Use test coverage tools like coverage.py to evaluate which code paths are exercised.

## 🧱 5.5 Introducing Logging and Observability

Logs help you:

* Trace what happened before an error.
* Understand user behavior.
* Audit sensitive actions.

**📦 Add Python logging:**

python

import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s %(message)s')

@app.route('/book/<int:book\_id>')

def get\_book(book\_id):

logging.info(f"Fetching book ID: {book\_id}")

...

Consider logging levels:

* DEBUG
* INFO
* WARNING
* ERROR
* CRITICAL

🔍 Suggestion: Log request method, path, headers (cautiously), and client IP for traceability.

## 📡 5.6 External Monitoring Tools

In real-world environments, use tools like:

| **Tool** | **Purpose** |
| --- | --- |
| **Prometheus + Grafana** | Real-time metrics dashboard |
| **ELK Stack** | Centralized logging (ElasticSearch, Logstash, Kibana) |
| **Datadog / New Relic** | Application performance monitoring |
| **Sentry / Rollbar** | Error tracking and alerting |

🧠 Educator Angle: Introduce learners to logs vs metrics. Logs show discrete events; metrics show trends.

## 🧪 5.7 Rate Limits, Timeouts, and Error Recovery

Robust APIs should:

* **Limit request rate** to avoid denial-of-service
* **Timeout gracefully** during long operations
* **Handle retries** with idempotency on POST

Example rate limit header:

http

X-RateLimit-Limit: 1000

X-RateLimit-Remaining: 998

X-RateLimit-Reset: 1442005983

🔐 Challenge: Create middleware to enforce request-per-IP limits with exponential backoff.

## 📖 5.8 Writing Tests for Edge Cases

Encourage students to cover:

* Malformed payloads
* Missing fields
* Invalid tokens
* Duplicate entries
* Empty states (GET /book when no books exist)

🔬 Teach defensive testing: not just what works, but what *shouldn’t* work.

## 🧠 Summary and Transition

This chapter taught learners how to treat APIs like production systems—not fragile demo code. They now know how to validate, observe, and test their endpoints consistently.

In **Chapter 6**, we take integration to the next level. You’ll build a front-end consumer for your API (HTML or React), stitch together REST and SOAP services, and prepare the system for real-world deployment.

# 🧩 Chapter 6: Integrating APIs Into Real Applications

## 📘 6.1 Integration Isn’t Just Connection—It’s Coordination

Just calling an endpoint isn’t integration. Integration means:

* Translating data formats
* Coordinating sequential requests
* Handling failures and timeouts
* Exposing orchestration logs for traceability

🔍 Analogy: You can call the pizza place, but full integration means ordering, paying, tracking delivery, and alerting if it’s late.

## 🛠️ 6.2 Building a Front-End Interface

Let’s start with a simple HTML form that consumes the Book Info API.

**📄 HTML + JavaScript Client**

html

<!-- file: book\_ui.html -->

<!DOCTYPE html>

<html>

<head><title>Book Lookup</title></head>

<body>

<h1>Find a Book</h1>

<input type="number" id="bookId" placeholder="Enter Book ID"/>

<button onclick="getBook()">Search</button>

<pre id="output"></pre>

<script>

function getBook() {

const id = document.getElementById('bookId').value;

fetch(`http://localhost:5000/book/${id}`)

.then(res => res.json())

.then(data => {

document.getElementById('output').textContent = JSON.stringify(data, null, 2);

})

.catch(err => console.error("Error:", err));

}

</script>

</body>

</html>

🧠 Learner Tip: This teaches asynchronous fetch and DOM interaction basics. Easily extensible to React or Vue later.

## 🛠️ 6.3 Orchestrating Multiple REST Calls

**⚙️ Use Case: Feedback Aggregator**

Let’s say users post feedback on books, and we want to:

1. Retrieve book info
2. Submit feedback
3. Fetch sentiment analysis

🧱 Example Service Flow:

python

def analyze\_feedback(book\_id, comment):

book = requests.get(f"http://localhost:5000/book/{book\_id}").json()

feedback\_payload = {"book\_id": book\_id, "comment": comment}

requests.post("http://localhost:5000/feedback", json=feedback\_payload)

sentiment = requests.get(f"http://localhost:5000/sentiment?text={comment}").json()

return {

"title": book['title'],

"feedback": comment,

"sentiment": sentiment['score']

}

💡 Tip: Teach response caching, error handling, and timeout strategies.

## 🌐 6.4 Integrating a SOAP Service Into REST Workflow

Let’s say book metadata comes from a legacy SOAP system. We'll proxy it through REST.

**🔄 Workflow:**

User → React UI → REST API → SOAP Broker → XML Payload → JSON Response

**🔧 Python SOAP Proxy:**

python

from flask import Flask, jsonify

from zeep import Client

soap\_client = Client("http://legacy-system.com/book.wsdl")

@app.route('/book/<int:book\_id>', methods=['GET'])

def get\_book(book\_id):

response = soap\_client.service.GetBookInfo(book\_id)

book\_json = {

"title": response.Title,

"author": response.Author,

"year": response.Year

}

return jsonify(book\_json)

🧠 Educator Insight: This demystifies SOAP for REST-native learners by embedding it inside REST routes.

## 📦 6.5 Versioning and Maintainability

Always version your APIs:

plaintext

/api/v1/book

/api/v2/book # adds genre, ratings

Versioning protects downstream clients from breaking changes and enables iterative design.

**🔧 Flask Blueprint:**

python

from flask import Blueprint

v1 = Blueprint('v1', \_\_name\_\_, url\_prefix='/api/v1')

@v1.route('/book/<int:id>')

def get\_book\_v1(id): ...

## 🧪 6.6 Documentation and Integration Help

Use tools from Chapter 3:

* Swagger for REST
* WSDL for SOAP
* Postman for collections
* README.md for command-line clients

Encourage learners to build a **developer onboarding page**:

* How to call each endpoint
* Sample curl commands
* Expected JSON shapes
* Authentication notes

## 🎓 6.7 Bringing It All Together: Customer Feedback Analyzer

**System Includes:**

* Front-end form to submit book feedback
* REST proxy to SOAP service for book info
* REST endpoints for feedback and sentiment
* Secure API with JWT
* Swagger UI for documentation
* Logging and health checks (Chapter 5)

🔗 Bonus: Host on Replit, Render, or GitHub Pages for learners to see it work end-to-end.

🧠 Summary and What’s Next

This chapter shows integration in action—not just API calls, but workflows. Next comes the appendix: cheat sheets, glossary, and curated resources to help learners extend this into HL7 messaging, Azure Functions, or production deployments.

# 📚 Appendix: Cheat Sheets, Glossary, and Further Reading

## 🗂️ REST & SOAP Command Cheat Sheets

**🔧 Common REST Commands (with curl)**

bash

# GET book by ID

curl -X GET http://localhost:5000/book/1

# POST new book

curl -X POST -H "Content-Type: application/json" \

-d '{"title":"Domain-Driven Design","author":"Eric Evans","year":2003}' \

http://localhost:5000/book

# PUT to update a book

curl -X PUT -H "Content-Type: application/json" \

-d '{"year":2004}' http://localhost:5000/book/1

# DELETE book

curl -X DELETE <http://localhost:5000/book/1>

## 🧼 Common SOAP Elements

xml

<soap:Envelope>

<soap:Header>

<wsse:Security>

<wsse:UsernameToken>

<wsse:Username>user</wsse:Username>

<wsse:Password>pass</wsse:Password>

</wsse:UsernameToken>

</wsse:Security>

</soap:Header>

<soap:Body>

<GetBookInfo>

<BookID>1</BookID>

</GetBookInfo>

</soap:Body>

</soap:Envelope>

## 🗣️ Glossary of API Terms

| **Term** | **Meaning** |
| --- | --- |
| API | Application Programming Interface — system-to-system communication tool |
| Endpoint | URL that represents a resource or action |
| REST | Representational State Transfer — HTTP-based interface style |
| SOAP | XML-based protocol for structured messaging |
| WSDL | Web Services Description Language — describes SOAP services |
| HTTP Method | Verb like GET, POST, PUT, DELETE |
| JWT | JSON Web Token — stateless authentication tool |
| OAuth2 | Authorization protocol for delegated access |
| Status Code | HTTP integer code — e.g., 200 (OK), 404 (Not Found) |
| Serialization | Converting objects to strings or byte formats (e.g. JSON, XML) |
| CORS | Cross-Origin Resource Sharing — browser-based access policy |

**🧭 Suggested Learning Paths**

**💻 Beginner Path**

* Build REST APIs using Flask or Express.js
* Use Postman to test endpoints manually
* Learn basic status codes and request/response cycles
* Read: “APIs for Dummies” by Mark Masse

**👨‍💻 Intermediate Path**

* Implement OAuth2 and JWT authentication
* Learn automated testing with pytest or unittest
* Integrate Swagger/OpenAPI and SOAP via proxy
* Read: “Designing Web APIs” by Brenda Jin

**🧠 Advanced Path**

* Explore microservices orchestration patterns
* Study message brokers (RabbitMQ, Kafka)
* Deep-dive HL7 v2 and v3 messaging structure in healthcare
* Read: “Enterprise Integration Patterns” by Hohpe & Woolf and HL7 Standard documentation (hl7.org)

**🔗 Curated Resources**

| **Topic** | **Recommended Source** |
| --- | --- |
| REST Fundamentals | restfulapi.net |
| SOAP Examples | TutorialsPoint |
| JWT Playground | jwt.io |
| Postman Docs | Learning Center |
| Swagger (OpenAPI) | Swagger.io |
| HL7 Interoperability | HL7 Education |
| Secure API Design | OWASP API Security Top 10 |
| Python API Dev | Real Python’s REST API articles (realpython.com) |
| Flask Documentation | Flask Docs |
| C# Web API Documentation | ASP.NET Docs |

# 💾 Full Code Suite: Building & Using APIs and SOAP Messages

## 📁 Project Structure Overview

booklet-project/

├── book\_api.py # Chapter 1: Basic Flask API

├── book\_client.py # Chapter 1: Python client for API

├── book\_api\_crud.py # Chapter 2: Full CRUD API

├── book\_client\_crud.py # Chapter 2: Expanded client

├── book\_api\_swagger.py # Chapter 3: Swagger documented API

├── book\_client\_test.py # Chapter 3: Automated tests

├── book\_api\_secure.py # Chapter 4: API Key + JWT security

├── book\_client\_auth.py # Chapter 4: Authenticated requests

├── soap\_proxy.py # Chapter 6: REST proxy to SOAP

├── feedback\_orchestrator.py # Chapter 6: Multi-endpoint flow

├── static/

│ └── book\_ui.html # Chapter 6: Front-end browser client

└── tests/

└── test\_book\_api.py # Chapter 5: Pytest test cases

## 📘 Chapter 1: Basic REST API and Client

**🔧 book\_api.py**

python

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

BOOKS = {

1: {"title": "The Pragmatic Programmer", "author": "Andrew Hunt", "year": 1999},

2: {"title": "Clean Code", "author": "Robert C. Martin", "year": 2008},

3: {"title": "Fluent Python", "author": "Luciano Ramalho", "year": 2015}

}

@app.route('/book/<int:book\_id>', methods=['GET'])

def get\_book(book\_id):

book = BOOKS.get(book\_id)

if book:

return jsonify(book)

return jsonify({"error": "Book not found"}), 404

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**🧪 book\_client.py**

python

import requests

def fetch\_book(book\_id):

url = f'http://127.0.0.1:5000/book/{book\_id}'

response = requests.get(url)

if response.status\_code == 200:

book = response.json()

print(f"Title: {book['title']}\nAuthor: {book['author']}\nYear: {book['year']}")

else:

print("Error:", response.json().get('error'))

fetch\_book(2)

## 📘 Chapter 2: CRUD API with Client

**🔧 book\_api\_crud.py**

python

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

BOOKS = {

1: {"title": "The Pragmatic Programmer", "author": "Andrew Hunt", "year": 1999},

2: {"title": "Clean Code", "author": "Robert C. Martin", "year": 2008}

}

@app.route('/book/<int:book\_id>', methods=['GET'])

def get\_book(book\_id):

book = BOOKS.get(book\_id)

if book:

return jsonify(book)

return jsonify({"error": "Book not found"}), 404

@app.route('/book', methods=['POST'])

def add\_book():

data = request.json

new\_id = max(BOOKS) + 1

BOOKS[new\_id] = data

return jsonify({"message": "Book added", "book\_id": new\_id}), 201

@app.route('/book/<int:book\_id>', methods=['PUT'])

def update\_book(book\_id):

if book\_id not in BOOKS:

return jsonify({"error": "Not found"}), 404

BOOKS[book\_id].update(request.json)

return jsonify({"message": "Book updated", "book": BOOKS[book\_id]})

@app.route('/book/<int:book\_id>', methods=['DELETE'])

def delete\_book(book\_id):

if book\_id not in BOOKS:

return jsonify({"error": "Not found"}), 404

del BOOKS[book\_id]

return jsonify({"message": "Book deleted"})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**🧪 book\_client\_crud.py**

python

import requests

BASE = 'http://127.0.0.1:5000/book'

def create\_book():

res = requests.post(BASE, json={"title": "Fluent Python", "author": "Luciano Ramalho", "year": 2015})

print(res.json())

def read\_book():

res = requests.get(f"{BASE}/1")

print(res.json())

def update\_book():

res = requests.put(f"{BASE}/1", json={"year": 2020})

print(res.json())

def delete\_book():

res = requests.delete(f"{BASE}/1")

print(res.json())

create\_book()

read\_book()

update\_book()

delete\_book()

## 📘 Chapter 3: Swagger + Testing

**🔧 book\_api\_swagger.py**

python

from flask import Flask, jsonify

from flasgger import Swagger

app = Flask(\_\_name\_\_)

Swagger(app)

@app.route('/ping')

def ping():

"""Health check endpoint

---

responses:

200:

description: Alive

"""

return jsonify({"status": "ok"})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**🧪 test\_book\_api.py (in /tests)**

python

import requests

def test\_valid\_book():

res = requests.get("http://localhost:5000/book/1")

assert res.status\_code == 200

def test\_not\_found():

res = requests.get("http://localhost:5000/book/999")

assert res.status\_code == 404

## 📘 Chapter 4: Security with API Key + JWT

**🔐 book\_api\_secure.py**

python

from flask import Flask, request, jsonify

import jwt, datetime

app = Flask(\_\_name\_\_)

API\_KEY = "SECRET123"

SECRET = "TOKEN\_SECRET"

def require\_api\_key(f):

def wrap(\*args, \*\*kwargs):

if request.headers.get('X-API-KEY') != API\_KEY:

return jsonify({"error": "Unauthorized"}), 401

return f(\*args, \*\*kwargs)

wrap.\_\_name\_\_ = f.\_\_name\_\_

return wrap

@app.route('/secure-endpoint')

@require\_api\_key

def secure():

return jsonify({"message": "Success!"})

@app.route('/login', methods=['POST'])

def login():

token = jwt.encode({"user": "randy", "exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1)}, SECRET, algorithm="HS256")

return jsonify({"token": token})

@app.route('/protected')

def protected():

token = request.headers.get('Authorization')

try:

jwt.decode(token, SECRET, algorithms=["HS256"])

return jsonify({"message": "Token valid!"})

except Exception as e:

return jsonify({"error": str(e)}), 403

app.run(debug=True)

## 📘 Chapter 6: Orchestration + SOAP Proxy + UI

**🔄 feedback\_orchestrator.py**

python

import requests

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

@app.route('/analyze', methods=['POST'])

def analyze():

data = request.json

book = requests.get(f"http://localhost:5000/book/{data['book\_id']}").json()

sentiment = requests.get(f"http://localhost:5000/sentiment?text={data['comment']}").json()

return {

"title": book['title'],

"feedback": data['comment'],

"sentiment": sentiment['score']

}

app.run(debug=True)

**🧼 soap\_proxy.py**

python

from flask import Flask, jsonify

from zeep import Client

app = Flask(\_\_name\_\_)

soap\_client = Client("http://legacy.com/wsdl")

@app.route('/book/<int:id>')

def get\_book(id):

book = soap\_client.service.GetBookInfo(id)

return jsonify({

"title": book.Title,

"author": book.Author,

"year": book.Year

})

app.run(debug=True)

**🌐 book\_ui.html**

html

<!DOCTYPE html>

<html>

<head><title>Book Lookup</title></head>

<body>

<input id="bookId" type="number" />

<button onclick="getBook()">Fetch</button>

<pre id="result"></pre>

<script>

function getBook() {

const id = document.getElementById('bookId').value;

fetch(`http://localhost:5000/book/${id}`)

.then(res => res.json())

.then(data => {

document.getElementById('result').textContent = JSON.stringify(data, null, 2);

});

}

</script>

</body>

</html>