Building REST APIs with Flask

Create Python Web Services with MySQL

Kunal Relan

Apress®

Building REST APIs with Flask

Create Python Web Services with MySQL

Kunal Relan

Apress®

Building REST APIs with Flask: Create Python Web Services with MySQL

Kunal Relan
New Delhi, Delhi, India

ISBN-13 (pbk): 978-1-4842-5021-1 ISBN-13 (electronic): 978-1-4842-5022-8
https://doi.org/10.1007/978-1-4842-5022-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: Laura Berendson
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

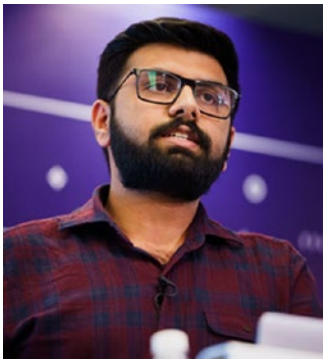
Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484250211. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to caffeine and sugar, my companions through many long night of writing, and extra credits to my mom.

Table of Contents

About the Author



Kunal Relan is an iOS security researcher and a full stack developer with more than four years of experience in various fields of technology, including network security, DevOps, cloud infrastructure, and application development, working as a consultant with start-ups around the globe. He is an Alibaba Cloud MVP and author of *iOS Penetration Testing* (Apress) and a variety of white papers. Kunal is a technology enthusiast and an active speaker. He regularly contributes to open source communities and writes articles for Digital Ocean and Alibaba Techshare.

About the Technical Reviewer

Saurabh Badhwar is a software engineer with a passion to build scalable distributed systems. He is mostly working to solve challenges related to performance of software at a large scale and has been involved in building solutions that help other developers quickly analyze and compare performance of their systems when running at scale.
He is also passionate about working with open source communities and has been actively participating as a contributor in various domains, which involve development, testing, and community engagement. Saurabh has also been an active speaker at various conferences where he has been talking about performance of large-scale systems.

Acknowledgments

I would like to thank Apress for providing me this platform, without which this would have been a lot harder. I would also like to thank Mr. Nikhil Karkal for his help and Miss Divya Modi for her perseverance, without whom this would have been a farsighted project.

I'd like to mention about the strong Python community which helped me understand the core concepts in my early years of programming, which inspired me to contribute back to the community with this book.

Last but certainly not the least, I would like to acknowledge all the people who constantly reminded me about the deadlines and helped me write this book, especially my family and Aparna Abhijit for helping me out with editing.

Introduction

Flask is a lightweight microframework for web applications built on top of Python, which provides an efficient framework for building web-based applications using the flexibility of Python and strong community support with the capability of scaling to serve millions of users.

Flask has excellent community support, documentation, and supporting libraries; it was developed to provide a barebone framework for developers, giving them the freedom to build their applications using their preferred set of libraries and tools.

This book takes you through different stages of a REST API–based application development process using flask which explains the basics of the Flask framework assuming the readers understand Python. We'll cover database integration, understanding REST services, REST APIs performing CRUD operations, user authentication, third-party library integrations, testing, deployment, and application monitoring.

At the end of this book, you'll have a fair understanding of Flask framework, REST, testing, deploying, and managing Flask applications, which will open doors to understanding REST API development.

CHAPTER 1

Beginning with Flask

Flask is a BSD licensed, Python microframework based on Werkzeug and Jinja2. Being a microframework doesn't make it any less functional; Flask is a very simple yet highly extensible framework. This gives developers the power to choose the configuration they want, thereby making writing applications or plugins easy. Flask was originally created by Pocoo, a team of open source developers in 2010, and it is now developed and maintained by The Pallets Project who power all the components behind Flask. Flask is supported by an active and helpful developer community including an active IRC channel and a mailing list.

Introduction to Flask

Flask has two major components, Werkzeug and Jinja2. While Werkzeug is responsible for providing routing, debugging, and Web Server Gateway Interface (WSGI), Flask leverages Jinja2 as template engine. Natively, Flask doesn't support database access, user authentication, or any other high-level utility, but it does provide support for extensions integration to add all such functionalities, making Flask a micro- yet production-ready framework for developing web applications and services. A simple Flask application can fit into a single Python file or it can be modularized to create a production-ready application. The idea behind Flask is to build a good foundation for all applications leaving everything else on extensions.

K. Relan, *Building REST APIs with Flask*, https://doi.org/10.1007/978-1-4842-5022-8_1

Flask community is quite big and active with hundreds of open source extensions. The Flask core team continuously reviews extensions and ensures approved extensions are compatible with the future releases. Flask being a microframework provides flexibility to the developers to choose the design decisions appropriate to their project. It maintains a registry of extensions which is regularly updated and continuously maintained.

Starting Flask

Flask, just like all other Python libraries, is installable from the Python Package Index (PPI) and is really easy to setup and start developing with, and it only takes a few minutes to getting started with Flask. To be able to follow this book, you should be familiar with Python, command line (or at least PIP), and MySQL.

As promised, Flask is really easy to start with, and just five lines of code lets you get started with a minimal Flask application.

Listing 1-1. Basic Flask Application

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, From Flask!'

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

The preceding code imports the Flask library, initiates the application by creating an instance of the Flask class, declares the route, and then defines the function to execute when the route is called. This code is enough to start your first Flask application.

The following code launches a very simple built-in server, which is good enough for testing but probably not when you want to go in production, but we will cover that in the later chapters.

When this application starts, the index route upon request shall return "Hello From Flask!" as shown in Figure 1-1.

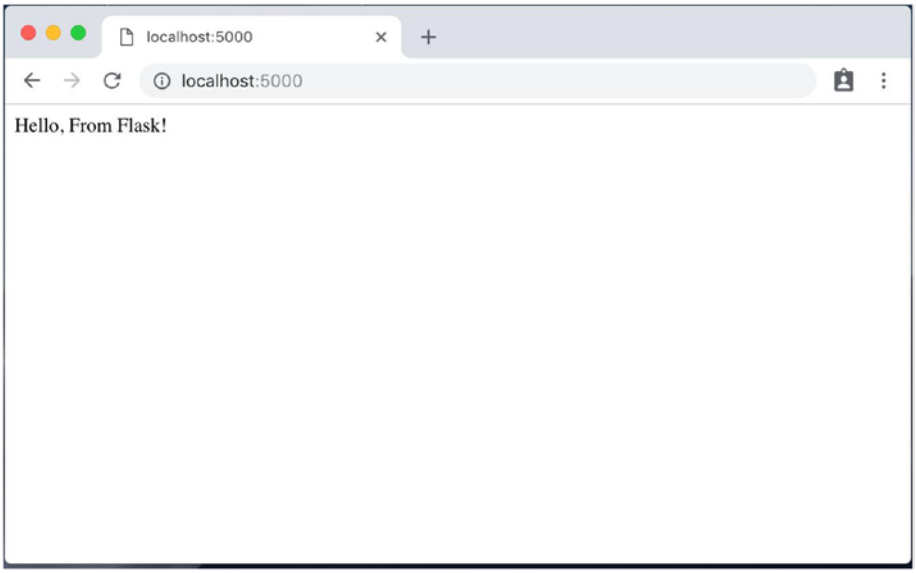


Figure 1-1. *Flask minimal application*

Flask Components Covered in This Book

Now that you have been introduced to Flask, we will discuss the components that we'll cover in Flask REST API development in this book.

This book will serve as a practical guide to REST API development using Flask, and we'll be using MySQL as the backend database. As already discussed, Flask doesn't come with native database access support, and to bridge that gap, we'll use a Flask extension called Flask-SQLAlchemy which adds support for SQLAlchemy in Flask. SQLAlchemy is essentially

a Python SQL toolkit and Object Relational Mapper which provides the developers the full power and flexibility of SQL.

SQLAlchemy provides full support for enterprise-level design patterns and is designed for high-performing database access while maintaining efficiency and ease of use. We'll build a user authentication module, CRUD (Create, Read, Update, and Delete) REST APIs for object creation, retrieval, manipulation, and deletion. We'll also integrate a documentation utility called Swagger for creating API documentation, write unit and integration tests, learn application debugging, and, finally, check out different methods of deploying and monitoring our REST APIs on cloud platforms for production use.

For unit tests, we'll use pytest which is a full-featured Python testing tool—pytest is easy to write tests with and yet is scalable to support complex use cases. We'll also use Postman which is a complete REST API Platform—Postman provides integration tools for every stage of the API lifecycle, making API development easier and more reliable.

API deployment and monitoring are critical parts of REST API development; development paradigm changes drastically when it comes to scaling the APIs for production use cases, and for the sake of this book, we'll deploy our REST APIs using uWSGI and Nginx on a cloud Ubuntu server. We'll also deploy our REST APIs on Heroku which is a cloud platform that facilitates Flask app deployment and scaling out of the box.

Last but not least, we'll discuss debugging common Flask errors and warnings and debugging Nginx requests and check out Flask application monitoring ensuring least amount on the downtime for production use.

Introduction to RESTful Services

Representational State Transfer (REST) is a software architectural style for web services that provides a standard for data communication between different kinds of systems. Web services are open standard

web applications that interact with other applications with a motive of exchanging data making it an essential part of client server architecture in modern web and mobile applications. In simple terms, REST is a standard for exchanging data over the Web for the sake of interoperability between computer systems. Web services which conform to the REST architectural style are called RESTful web services which allow requesting systems to access and manipulate the data using a uniform and predefined set of stateless operations.

Since its inception in 2000 by Roy Feilding, RESTful architecture has grown a lot and has been implemented in millions of systems since then. REST has now become one of the most important technologies for web-based applications and is likely to grow even more with its integration in mobile and IoT-based applications as well. Every major development language has frameworks for building REST web services. REST principles are what makes it popular and heavily used. REST is stateless, making it straightforward for any kind of system to use and also making it possible for each request to be served by a different system.

REST enables us to distinguish between the client and the server, letting us implement the client and the server independently. The most important feature of REST is its statelessness, which simply means that neither the client nor the server has to know the state of each other to be able to communicate. In this way, both the client and the server can understand any message received without seeing the previous message. Since we are talking about RESTful web services, let's take a dive into web services and compare other web service standards.

Web services in a simple definition is a service offered by one electronic device to another, enabling the communication via the World Wide Web. In practice, web services provide resource-oriented, web-based interface to a database server and so on utilized by another web client. Web services provide a platform for different kinds of systems to communicate to each other, using a solution for programs to be able to communicate with each other in a language they understand (Figure 1-2).

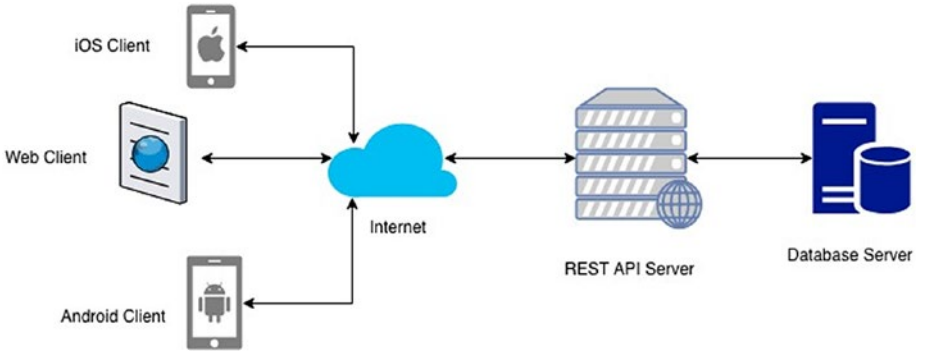


Figure 1-2. *REST architecture diagram*

SOAP (Simple Object Access Protocol) is another web service communication protocol which has been overtaken by REST in the recent years. REST services now dominate the industry representing more than 70% of public APIs according to Stormpath. They operate by exposing consistent interface to access named resources. SOAP, however, exposes components of application logic as services rather than data. SOAP is now a legacy protocol originally created by Microsoft and has a lot of other constraints when compared to REST. SOAP only exchanges data over XML, and REST provides the ability to exchange data over a variety of data formats. RESTful services are comparatively faster and less resource intensive. However, SOAP still has its own use cases in which it's a preferred protocol over REST.

SOAP is preferred when robust security is essential as it provides support for Web Services Security (WS-Security), which is a specification defining how security measures are implemented in web services to protect them from external attacks. Another advantage of SOAP over REST is its built-in retry logic to compensate for failed requests unlike REST in which the client has to handle failed requests by retrying. SOAP is highly extensible with other technologies and protocols like WS-Security, WS-addressing, WS-coordination, and so on which provides it an edge over other web service protocols.

Now, when we have briefly discussed web services—REST and SOAP—let's discuss features of REST protocol. In general, REST services are defined and implemented using the following features:

1. Uniform interface
2. Representations
3. Messages
4. Links between resources
5. Caching
6. Stateless

Uniform Interface

RESTful services should have a uniform interface to access resources, and as the name suggests, APIs' interface for the system should be uniform across the system. A logical URI system with uniform ways to fetch and manipulate data is what makes REST easy to work with. HTTP/1.1 provides a set of methods to work on noun-based resources; the methods are generally called verbs for this purpose.

In REST architecture, there is a concept of safe and idempotent methods. Safe methods are the ones that do not modify resources like a GET or a HEAD method. An idempotent method is a method which produces the same result no matter how many times it is executed. Table 1-1 provides a list of commonly used HTTP verbs in RESTful services.

Table 1-1. *Commonly used HTTP verbs useful in RESTful services*

Verb	CRUD	Operation	Safe	Idempotent
GET	Read	Fetch a single or multiple resource	Yes	Yes
POST	Created	Insert a new resource	No	No
PUT	Update/ Create	Insert a new resource or update existing	No	Yes
DELETE	Delete	Delete a single or multiple resource	No	Yes
OPTIONS	READ	List allowed operations on a resource	Yes	Yes
HEAD	READ	Return only response headers and no body	Yes	Yes
PATCH	Update/ Modify	Only update the provided changes to the resource	No	No

Representations

RESTful services focus on resources and providing access to the resources. A resource can be easily thought of as an object in OOP. The first thing to do while designing RESTful services is identifying different resources and determining the relation between them. A representation is a machine-readable explanation defining the current state of a resource.

Once the resources are identified, representations are the next course of action. REST provides us the ability to use any format for representing the resources in the system. Unlike SOAP which restricts us to use XML to represent the data, we can either use JSON or XML. Usually, JSON is the preferred method for representing the resources to be called by mobile or web clients, but XML can be used to represent more complex resources.

Here is a small example of representing resources in both formats.

Listing 1-2. XML Representation of a Book Resource

```
<?xml version="1.0" encoding="UTF-8"?>
<Book>
  <ID> 1 </ID>
  <Name> Building REST APIs with Flask </Name>
  <Author> Kunal Relan </Author>
  <Publisher > Apress </ Publisher >
</Book>
```

Listing 1-3. JSON Representation of a Book resource

```
{
      "ID": "1",
      "Name": "Building REST APIs wiith Flask",
      "Author": "Kunal Relan",
      "Publisher": "Apress"
}
```

In REST Systems, you can use either of the methods or both the methods depending on the requesting client to represent the data.

Messages

In REST architecture, which essentially established client–server style way of data communication, messages are an important key. The client and the server talk to each other via messages in which the client sends a message to the server which is often called as a request and the server sends a response. Apart from the actual data exchanged between the client and the server in the form of request and response body, there is some metadata exchanged by the client and the server both in the form of request and response headers. HTTP 1.1 defines request and response headers formats in the following way in order to achieve a uniform way of data communication across different kinds of systems (Figure 1-3).

Figure 1-3. *HTTP sample request*

In Figure 1-4, GET is the request method, "/comments" is the path in the server, "postId=1" is a request parameter, "HTTP/1.1" is the protocol version that the client is requesting, "jsonplaceholder.typicode.com" is the server host, and content type is a part of the request headers. All of these combined is what makes a HTTP request that the server understands.

In return, the HTTP server sends the response for the requested resources.

```
[
  {
    "postId": 1,
    "id": 1,
    "name": "id labore ex et quam laborum",
    "email": "Eliseo@gardner.biz",
    "body": "laudantium enim quasi est quidem magnam voluptate
            ipsam eos\ntempora quo necessitatibus\ndolor quam
            autem quasi\nreiciendis et nam sapiente accusantium"
  },
  {
    "postId": 1,
    "id": 2,
    "name": "quo vero reiciendis velit similique earum",
    "email": "Jayne_Kuhic@sydney.com",
```