



# **Introduction to API Development Using Python**

## **Participant Guide**



#### Copyright

This subject matter contained herein is covered by a  
copyright owned by: Copyright © 2023 Robert Gance, LLC

This document contains information that may be proprietary. The contents of this document may not be duplicated by any means without the written permission of TEKsystems.

TEKsystems, Inc. is an Allegis Group, Inc. company. Certain names, products, and services listed in this document are trademarks, registered trademarks, or service marks of their respective companies.

All rights reserved

7437 Race Road  
Hanover, MD 21076

COURSE CODE IN1803 / 11.1.2022  
Phase 4 Accessibility / 8.10.2022

©2023 Robert Gance, LLC

ALL RIGHTS RESERVED

This course covers Introduction to API Development Using Python

---

No part of this manual may be copied, photocopied, or reproduced in any form or by any means without permission in writing from the author—Robert Gance, LLC, all other trademarks, service marks, products or services are trademarks or registered trademarks of their respective holders.

This course and all materials supplied to the student are designed to familiarize the student with the operation of the software programs.

THERE ARE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, MADE WITH RESPECT TO THESE MATERIALS OR ANY OTHER INFORMATION PROVIDED TO THE STUDENT. ANY SIMILARITIES BETWEEN FICTITIOUS COMPANIES, THEIR DOMAIN NAMES, OR PERSONS WITH REAL COMPANIES OR PERSONS IS PURELY COINCIDENTAL AND IS NOT INTENDED TO PROMOTE, ENDORSE, OR REFER TO SUCH EXISTING COMPANIES OR PERSONS.

This version updated: 11/1/2022

# *Notes*

---

# *Chapters at a Glance*

|           |                                  |     |
|-----------|----------------------------------|-----|
| Chapter 1 | Introduction and Overview        | 14  |
| Chapter 2 | Fundamentals                     | 45  |
| Chapter 3 | Advanced Concepts                | 96  |
| Chapter 4 | Authorization and Authentication | 115 |
|           | Course Summary                   | 128 |
| Appendix  | Python Primer                    | 133 |

# *Notes*

---

# Introduction to API Development Using Python

## Course Objectives

- Understand key API development principles
- Develop and test RESTful APIs
- Examine additional capabilities such as caching, versioning, pagination, security

# Course Agenda

## Day 1

### Introduction

Overview of API Concepts

Python Tools and Environment Setup

### Developing APIs

Creating an API

Creating a Python-Based Client

Using Flask-RESTX

# Course Agenda *(continued)*

## **Day 2**

### Developing APIs *(continued)*

Incorporating a Database

Advanced Concepts

Pagination

Versioning

Caching

Authorization and Authentication

## Introductions

Name (prefer to be called) 

What you work on 

Background / experience with Python 

Reason for attending 

## Typical Daily Schedule\*

|              |                   |
|--------------|-------------------|
| 9:00         | Start Day         |
| 10:10        | Morning Break 1   |
| 11:20        | Morning Break 2   |
| 12:30 – 1:30 | Lunch             |
| 2:40         | Afternoon Break 1 |
| 3:50         | Afternoon Break 2 |
| 5:00         | End of Day        |

\* Your schedule may vary, timing is approximate

## Get the Most From Your Experience



Ask Questions

# Chapter 1

# Introduction and Overview



# Chapter 1 Overview

## **API Concepts and Overview**

API Key Components and Life Cycle

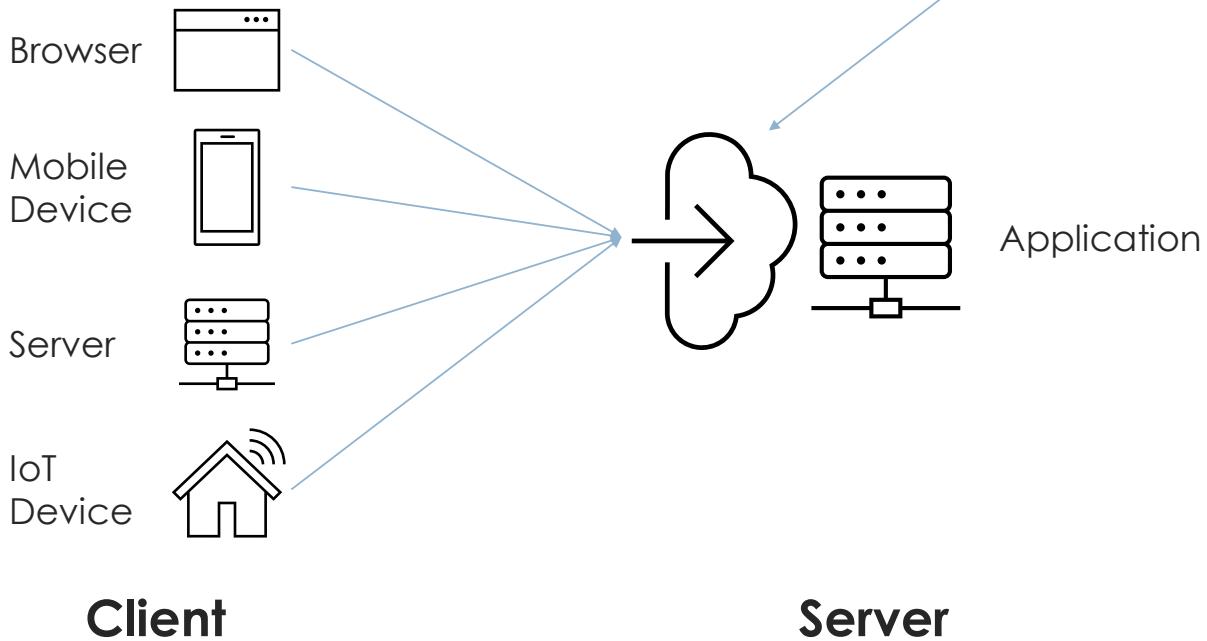
Service Architectures

Setting Up the Python Environment

Tools for API Development

# What Is an API?

- API stands for Application Programming Interface



API is an acronym that stands for Application Programming Interface. The acronym, usually used as a noun (e.g., "an API"), refers to the way in which one application communicates with another application. These applications are usually running on different machines.

# Modern Types of APIs

- API stands for Application Programming Interface

**The application server provides access to its data or services in one of two ways:**

**Web Services**

**RESTful Services**

EJB

XML-RPC

CORBA

DCE/RPC

MSRPC

RMI

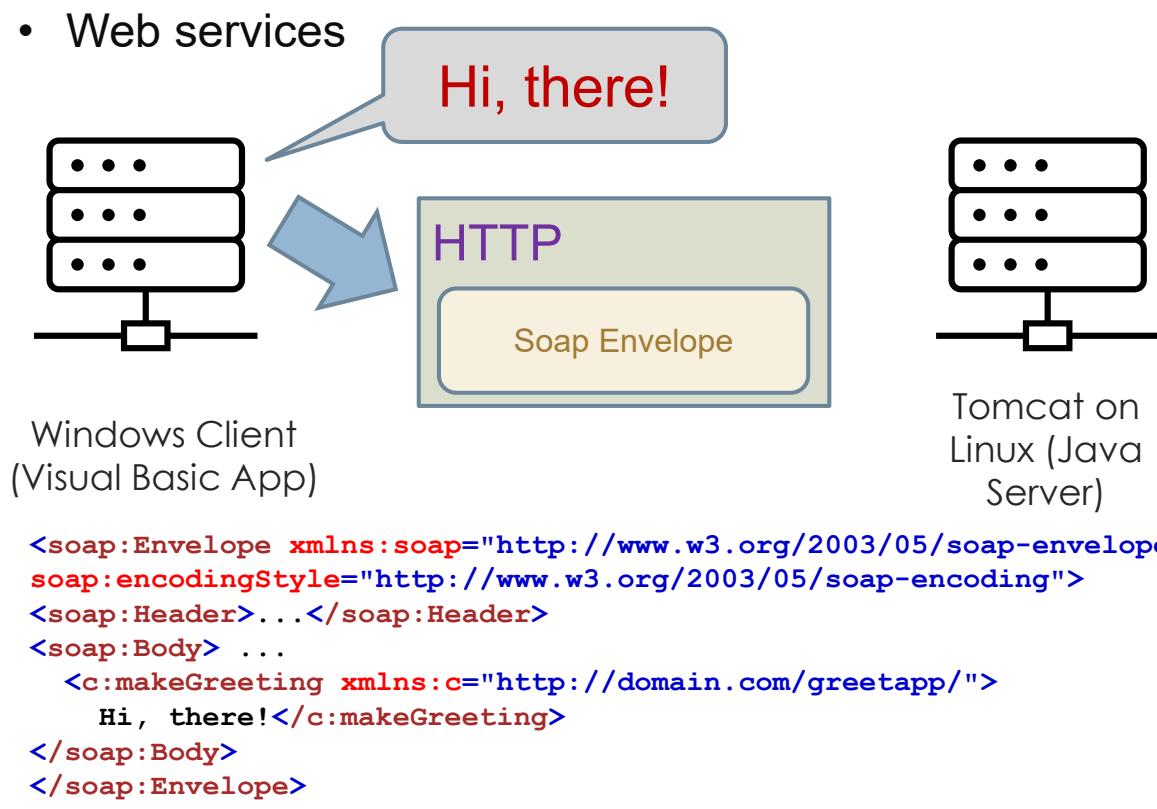
Over the years, different technologies for facilitating communication between two devices have come and gone (or largely been replaced). Many of these technologies focused on ways to transmit objects from one machine to another. They were often vendor-specific (focusing on the use of specific frameworks or tools) such as Microsoft DCOM or Java EJBs. Portability and interoperability were problems with these solutions. They were often complex and required creating special adapters called *stubs* and *skeletons* or required converting objects into other formats using serializers and deserializers. They were also very brittle requiring high maintenance.

As internet popularity grew in the 1990s, replacement technologies for distributed communications developed. Web services evolved due in part to the rise of XML at the time and the growing number of web-based servers that used the HTTP protocol.

Over time, web services would doom itself by becoming overly complicated (SOAP, WSDL, UDDI, WS-Security, WS-Reliability, etc.) and in response, the world shifted toward simpler technologies: JSON and REST.

# Web Services

- Web services



Web services evolved in an effort to improve interoperability between different systems. It emphasized the HTTP protocol as the transport layer and embedded XML as the payload within an HTTP POST request (commonly). Information could now easily be exchanged between a Microsoft Visual Basic Application running on Windows and a Java application server running on Linux or Unix.

## The Browser as a Client

- Browsers are good at running an embedded programming language called *JavaScript*
- The combined power of HTML, CSS, and JavaScript allowed browsers to become *thinner* clients
  - Older alternatives were called "thick" clients due to the more complex code required to run them
- Browsers, however, aren't very good at manipulating XML
- The lack of good XML processing capabilities within browsers spelled trouble for web services

During the late 1990s and early 2000s, browsers evolved rapidly. JavaScript became a predominant development language and gave rise to more interactive web applications that could incorporate new techniques like Ajax. This facilitated communication between the browser and the server. The early 2000s was a transformative period where web applications became powerful clients. The growth of JavaScript also gave rise to a special data format called JSON (pronounced “Jay-son”) which could easily be extracted and rendered into a web page.

# JSON

- **JavaScript Object Notation** (JSON) naturally occurs within the JavaScript language
  - Browsers consume and create JSON data without any additional tools
- Due to the quick rise in popularity of JSON, other programming languages quickly added support for it
- Within a few years, the JSON data format became a preferred way of exchanging data

JSON wasn't actually invented, it was discovered. It already existed within the JavaScript language. Doug Crockford (writer of the book *JavaScript: the Good Parts*) readily admits that he identified JSON, he didn't create it.

# The JSON Syntax

```
{
  "meals": [
    {
      "idMeal": "52775",
      "strMeal": "Vegan Lasagna",
      "strDrinkAlternate": null,
      "strCategory": "Vegan",
      "strArea": "Italian",
      "strInstructions": "Preheat oven to 180 degrees celcius...",
      "strTags": "Vegan,Pasta",
      "strYoutube": "https://www.youtube.com/watch?v=VU8cXvIGbvc",
      "strIngredient1": "green red lentils",
      "strIngredient2": "carrot",
      "strScalable": true,
      "strMeasure1": "1 cups",
      "strMeasure2": 1,
      "strMetric": false
    },
    {
      "idMeal": "52987",
      "strMeal": "Lasagna Sandwiches",
      "strCategory": "Pasta",
      "strArea": "American"
    }
  ]
}
```

**Object notation**

**Array notation**

**Objects are made of name / value pairs**

**Property names *MUST* be double (not single) quoted**

**Commas separate name / value pairs, except after the last pair in an object**

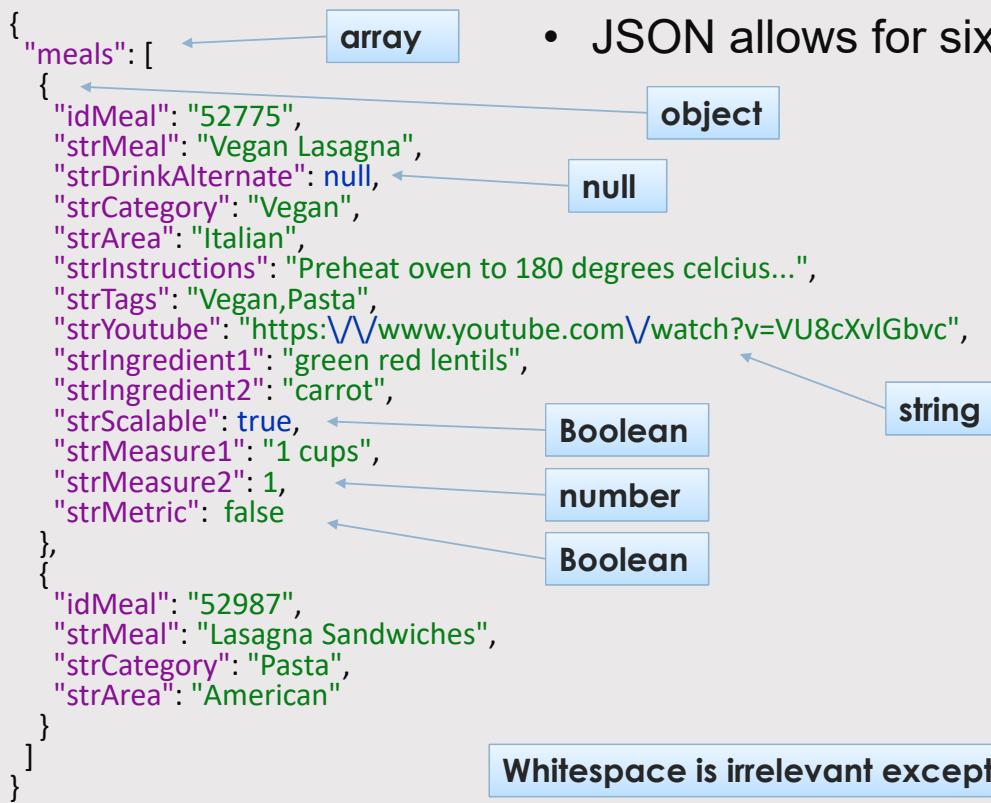
For many years, a standard for JSON did not exist. An RFC for JSON is managed by the IETF and can be found at: <https://datatracker.ietf.org/doc/html/rfc8259>.

Does the top-level structure of a JSON message have to be an object or an array? While it almost always is, recently a simple string, number, Boolean, or null is now allowed also.

This API response can easily be obtained by typing

<https://www.themealdb.com/api/json/v1/1/search.php?s=lasagna> into your browser.

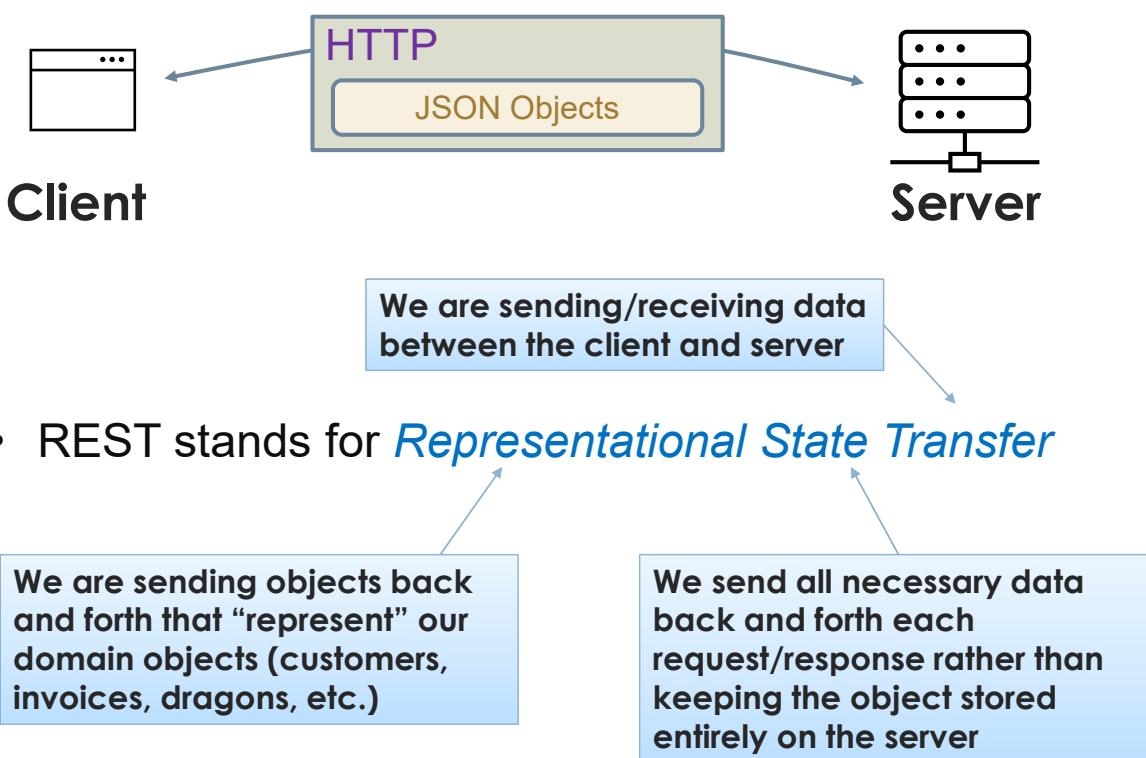
# Whitespace and JSON Data



- JSON allows for six data types

Valid JSON data types include objects, arrays, strings, numbers, true, false, and null. While JSON data is used within JavaScript, they are not the same. For example, JSON does not define a function type, an undefined type, or other types, such as a date type. Only the six types mentioned above are explicitly defined within JSON.

# RESTful Services



REST is merely a style of communication between two devices.

At first, the REST acronym seems rather cryptic, but upon dissecting it, we see it identifies how this the style of communication. REST involves sending objects to and from the server (the "transfer" part of the acronym). It often uses JSON (though it is not required) as the data interchange format. JSON objects "represent" data maintained on the server. The objects are sent back and forth each time (the "state" part of the acronym) rather than using cookies, for example, to uniquely identify the user and then bring up the objects in a server session like a typical web-based application.

# Classic Web Services vs RESTful APIs

| Web Services                                        | REST                                                      |
|-----------------------------------------------------|-----------------------------------------------------------|
| XML for payload data                                | Can use plain-text, XML, JSON (commonly), etc.            |
| Uses SOAP (Simple Object Access Protocol)           | No specific protocol                                      |
| Invokes methods remotely                            | Access data via URIs                                      |
| Has formal standards: SOAP, WSDL, WS-Security, etc. | Solutions tend to be simpler to learn, build, and execute |
| XML is difficult to work with within browsers       | JSON is easy to use within browsers                       |
| Has error handling capabilities built into SOAP     | Errors are managed through HTTP status codes              |

There are several differences between classic (SOAP-based) web services and RESTful solutions. One of the primary differences is that RESTful solutions utilize the HTTP protocol to manipulate resources while web services usually make HTTP POST requests to invoke methods on the server side. We'll discuss more about REST's manipulation of URLs shortly.

# RESTful Resources

- A **resource** is a core or fundamental component in a RESTful application
- A resource is an object being served (returned)
  - It usually derives from the business domain
  - Typically, operations will be performed on this entity
- Resource examples:
  - Invoices
  - Patients
  - Customers
  - Documents

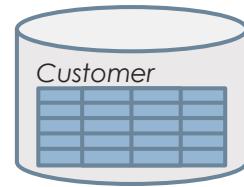
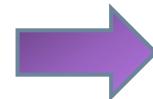
A resource can be a collection (like accounts) or singular (like account) and may contain sub-resources (like address)

Creating resources is a core task in designing RESTful solutions. Operations will be defined around these resources. Care must be taken on how large or small to make these resources. In other words, how coarse-grained vs. fine-grained the objects should be is a design decision that must be considered (more on resource modeling later). Larger resources will involve fewer requests made to the server but involve sending and receiving more data values.

# Key Components of a RESTful Service

- URLs to map to resources

`http://server.com/api/customers`



- Manipulation of resources via HTTP methods

GET

POST

PUT

DELETE

`http://server.com/api/customers`

- Transfer of state (stateless)



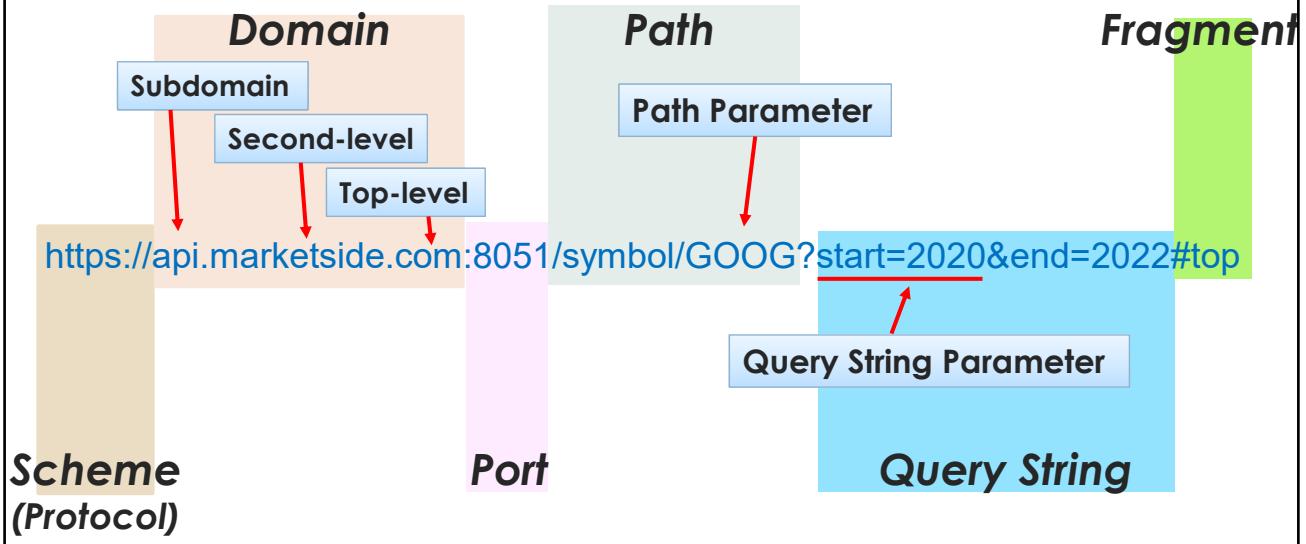
```
{  
    customer_id: 121,  
    name: Blackburn Industries,  
    ...  
}
```

- Cacheable

`Expires: Fri, 20 May 2022 19:20:49 GMT`

Several principles guide the development of RESTful solutions (each of these are discussed further later).

# Parts of a URL



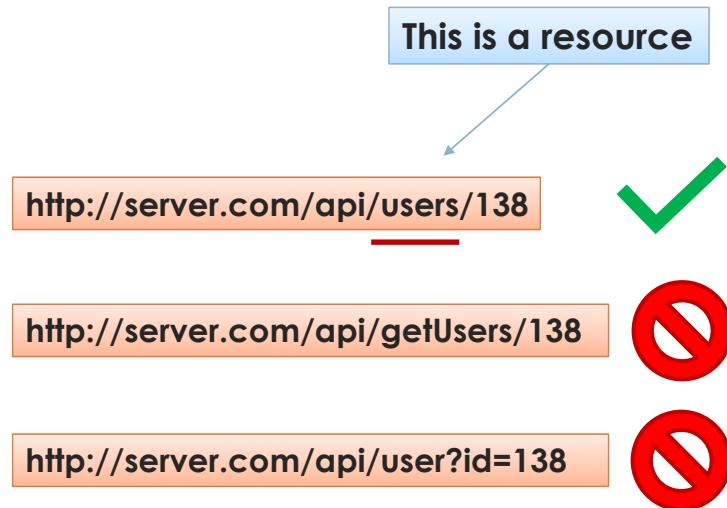
It will be important to distinguish between a path parameter and a query string parameter

For ease of discussion throughout the course, the terms above will be used to refer to parts of a URL.

It's important to take note of the difference in placement of variables in a URL. This distinction is not only a part of the RESTful design solution but makes a difference programmatically when writing code to access these values. Take note of what is called the "path" versus the "query string."

# RESTful Design Principles

- Design the API around resources (nouns)



Resources should map to a unique URI (like `http://server.com/api/users`). Resources are usually domain or business objects from your application.

To make the API sound more consistent, generally prefer the use of plural nouns.

# RESTful Design Principles (*continued*)

- Use HTTP methods to define operations

GET /api/invoices

Retrieves all invoices

GET /api/invoices/2952

Reads a specific invoice

POST /api/invoices

Creates a new invoice

PUT /api/invoices/2952

Updates a specific invoice

PATCH /api/invoices/2952

Partially Updates a specific invoice

DELETE /api/invoices/2952

Deletes a specific invoice

Think of a RESTful request as being like a sentence. The HTTP method is the "verb" of the sentence while the resource is the noun.

(from above) "GET (verb, HTTP method) the invoice 2952 (noun, resource)."

As previously mentioned, we tend to utilize the plural form of our resource in the URL.

The PATCH method is used when you only wish to update one or two (or so) fields of an object. In doing so, you may PATCH by only sending the required fields that need updating. Generally, a PUT will "resend" the entire object, performing an overwrite of the entire object, which may overwrite fields you didn't intend to update (and may accidentally change values that you didn't intend to change). More is discussed with PATCH a little later.

# HTTP/HTTPS

## HTTP

Request line

Headers

Message Body

**POST /process/state HTTP/1.1**

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64)

Host: www.sample.com

Content-Type: application/x-www-form-urlencoded

Content-Length: length

Accept-Language: en-us

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

state=Colorado&capital=Denver

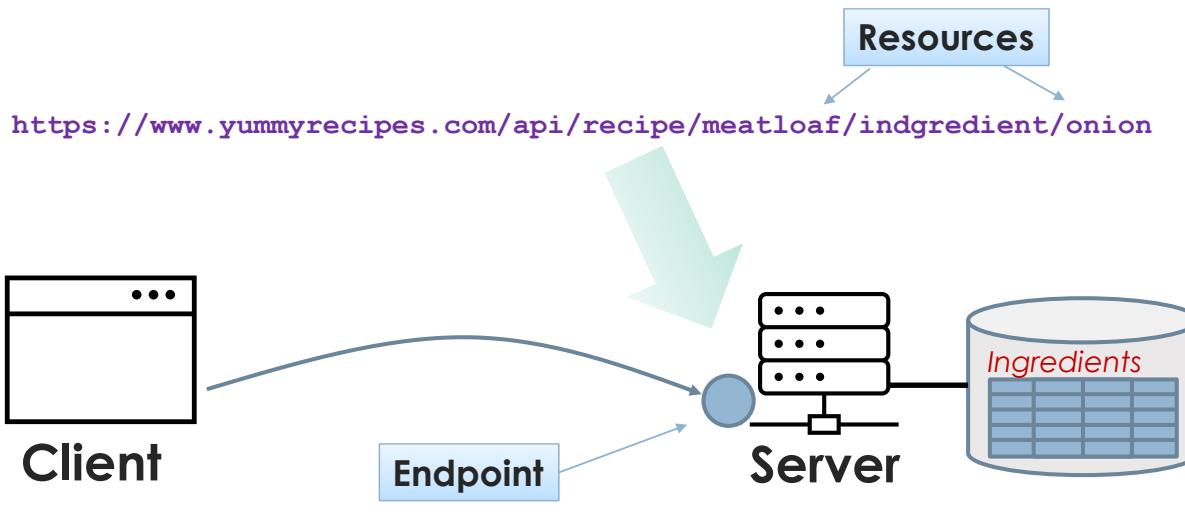
**POST, PUT, PATCH, and DELETE will place parameters to be sent in the HTTP message body**

To better understand how to set up and design an API, we'll have to understand the technologies used. The HTTP protocol is a good place to begin.

Above is a dissection of the HTTP protocol. The HTTP message defines a request line followed by zero or more headers followed by a blank line and then the message body. A GET request will not have a message body.

# Endpoints vs Resources

- Endpoints are *locations* (addresses), often exposing services or resources
- Resources are objects that are returned from a URL



You may often hear the term endpoint. Endpoints are typically addresses that identify a service location (commonly described in web services, but RESTful resource URIs could also be endpoints). Different endpoints can return similar resources. In addition, a single endpoint could return different resources such as collections of objects or single objects.

# Using Python for RESTful Development

- Python is one of the best languages for API development
  - One of the most popular languages to learn
  - Huge community support and tools to assist
  - Expressive language: easy to learn, easy to read
  - Works naturally with JSON data
    - Python dictionaries are very similar to the JSON format

Why Python for API development? We could specify a long list of why Python is so wonderful for RESTful development. But perhaps to help us get a better evaluation of whether Python would help, let's examine the question *why not Python?*

Here are a few disadvantages when considering Python for API development:

1. Performance. While Python can hold its own for most implementations, if high concurrency, high performance is a primary consideration, Python may not be the first choice.
2. Python on the client-side may be limited. Python isn't typically encountered frequently in mobile/handheld computing environments. However, as an option on the server-side it is a great choice.
3. Less developed database access layer. While wonderful tools do exist, like Django and SQLAlchemy, for interacting with the database, the layers sometimes feel less mature than in other languages such as Java and C# programming languages.

# RESTful Python API Frameworks

- Numerous tools exist for developing RESTful applications

## Server-side Tools

Django REST

Flask

Flask-RESTful

FastAPI

Falcon

Bottle

## Client-side Tools

Requests

Faster-than-requests

PycURL

This list of tools is not exhaustive. It includes useful frameworks and libraries that can be used on both the server-side as well as client-side. Several of these tools are explored during this course. The scope of Django is too large for this course.

## Your Turn! - Task 1-1

- Complete the setup of your Python environment
- *Follow provided instructions for Task 1-1 in the back of the student manual (participant guide)*

## Your Turn! - Task 1-2

- Viewing APIs
- *Use your browser to view results from the URLs*
- These can be easily obtained from **task1\_2\_starter.json**

<https://api.coinlore.net/api/tickers/?start1&limit=5>

Change these values, what happens?

<https://apimeme.com/meme?top=some&bottom=text>

Is this still an API? What does it return?

[https://datausa.io/api/data?  
drilldowns=State&measures=Population&year=latest](https://datausa.io/api/data?drilldowns=State&measures=Population&year=latest)

Have fun generating your own memes at <https://apimeme.com/>.

# Dictionaries

- Dictionaries are collections of name/value pairs

```
capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', ...,
            'Wyoming': 'Cheyenne'}
```

```
print(f'Keys: {capitals.keys()}' )
```

To get just the keys

Keys: dict\_keys(['Alabama', 'Alaska',  
... 'Wyoming'])

```
print(f'Values: {capitals.values()}' )
```

To get just the values

Values: dict\_values(['Montgomery',  
'Juneau', ..., 'Cheyenne'])

```
print(f'Keys and values: {capitals.items()}' )
```

To get both at once

Keys and values: dict\_items([('Alabama',  
'Montgomery'), ('Alaska', 'Juneau'), ...,  
('Wyoming', 'Cheyenne')])

```
print(f'{capitals["Montana"]}, {capitals.get("Montana")},  
{capitals.get("Montana", "Not found.")}' )
```

Helena, Helena, Helena

[student\\_files/ch01\\_overview/01\\_dicts\\_and\\_json.py](#)

Dictionaries are the default data structure encountered when going back and forth between JSON and Python. So, it is important to know to use them.

# The JSON Module

- Use `json.dumps(dict)` to convert from Python to JSON format

```
import json
```

To convert it to JSON

```
capitals_json = json.dumps(capitals)
print('To JSON:\n', capitals_json)
```

To JSON:

```
{"Alabama": "Montgomery",..., "Wyoming": "Cheyenne"}
```

- Use `json.loads(json_str)` to convert from JSON to a Python dict

```
capitals_dict = json.loads(capitals_json)
print('From JSON:\n', capitals_dict)
```

From JSON:

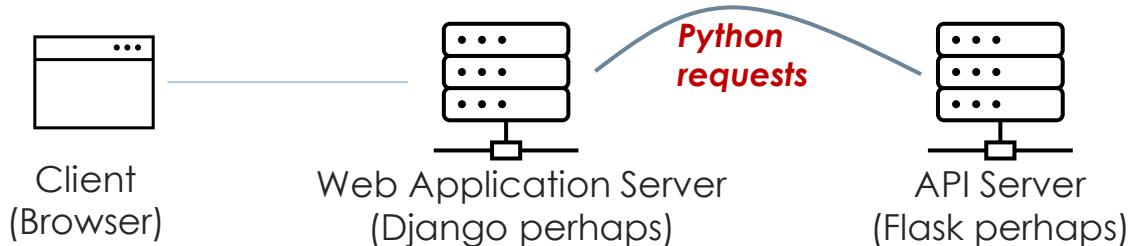
```
{'Alabama': 'Montgomery', ..., 'Wyoming': 'Cheyenne'}
```

*student\_files/ch01\_overview/01\_dicts\_and\_json.py*

The `json` module will help convert Python objects to JSON format. Objects will first need to be converted to dictionaries first. The global, built-in `vars()` function can assist with this.

# Introducing Python Requests

- While Python is not commonly used in mobile and handheld devices, it is heavily used on computing platforms (OS X, Windows, Linux)
- So, Python can also be used as a *RESTful client* in REST-based apps
- A great tool for building Python-based clients is called **requests**



Often, the client to an API server is another server. Quite frequently, we see a web-based server interacting with an API server.

In the Python world, this can be a Django frontend server interacting with a Flask API server (to be introduced shortly). The Django application would need to make requests to the remote API server. To do this, the `requests` module would need to be invoked from within the Django app.

# Python *requests*

- *requests* allows for easily sending HTTP requests
- *requests* is third-party, which means it must be installed into the Python environment
  - One of these statements likely works for you, however, if you already accomplished Task 1-2, you will already have *requests* installed

```
pip install requests  
pip3 install requests  
pip3.10 install requests  
python -m pip install requests
```

A basic *requests* request

```
text_str = requests.get(url,  
                        params = { param1: value1 },  
                        headers = { name: value }).text
```

**params** get added  
into the query string

As with any tool used within this course, if it doesn't ship with Python originally, it is considered a third-party tool and must be installed. The utility in Python for installing third-party frameworks is called *pip* (Python Installation Package tool).

Different Python versions on different platforms installed differently will likely need to invoke the pip tool in a slightly different way. You will need to remember how it was invoked on your system. Shown above are some variations on how to invoke the pip tool. If needed, discover the approach that is appropriate on your machine.

## Other *requests* Module APIs

```
python_dict = requests.get(url, params={param1: value1}).json()
```

Converts a JSON response  
to a Python dictionary

r = `requests.post(url,`

Python dict that ends  
up in the HTTP body

`params = { param1: value1 },`  
`data = { bodyParam1: bodyVal1 },`  
`json = obj,`  
`headers = { name: value }`

Alternatively, a Python object  
that ends up in the HTTP body

r = `requests.put()`

r = `requests.patch()`

r = `requests.delete()`

These each take similar  
arguments as `requests.post()`

The `requests` module methods shown here will serve as useful tools for creating a Python client shortly.

The **params** argument is useful for encoding dictionary key-value pairs into the query string of a URL.

The **data** argument is useful for encoding dictionary key-value pairs into the body of the HTTP message.

# Using the *requests* Module

```
import requests

r = requests.get('https://jsonplaceholder.typicode.com/posts/1', params={'age': 37})

print(r.url)          https://jsonplaceholder.typicode.com/posts/1?age=37

print(r.text)
    Raw text
    Dictionary
        print(r.json())
            {
                "userId": 1,
                "id": 1,
                "title": "sunt aut facere ... reprehenderit",
                "body": "quia et suscipit ... architecto"
            }
            {'userId': 1, 'id': 1, 'title': 'sunt aut ...  
reprehenderit', 'body': 'quia et suscipit ...  
architecto'}

print(r.headers)
    {'Date': 'Thu, 13 Jan 2022 02:39:07 GMT',
     'Content-Type': 'application/json', ...}

print(r.status_code)
    200
```

*student\_files/ch01\_overview/02\_using\_requests.py*

The code above makes a request to an API using the Python `requests` module. Various attributes are then displayed (shown in the green boxes) after the request completes.

## Your Turn! - Task 1-3

- Use the Python requests module to make a request and print the responses (either as raw text or as a dictionary) for the URLs used in Task 1-2
- *Work from the `task1_3_starter.py` file*

`https://api.coinlore.net/api/tickers/?start1&limit=5`  
`https://apimeme.com/meme?top=Tell%20Me%20More&bottom=About%20Chocolate`  
`https://datausa.io/api/data?drilldowns=State&measures=Population&year=latest`



Python-based Client

*Python  
requests*



Server

## Several Other API-Related Tools

- **Postman** – client-side interface for testing APIs 
- **Swagger** – open-source project that defines how to automatically generate documentation on APIs 
- **API Blueprint** – open-source documentation generation 
- **RAML** – design and model APIs using YAML syntax 

### API Management Tools

- Provide assistance in the design, integration, publishing, and securing of APIs
- Come at an increased cost
- Tools include:

 apigee

 MuleSoft

 SwaggerHub

Swagger emphasizes the use of JSON and YAML while API Blueprint emphasizes the use of markdown notation.

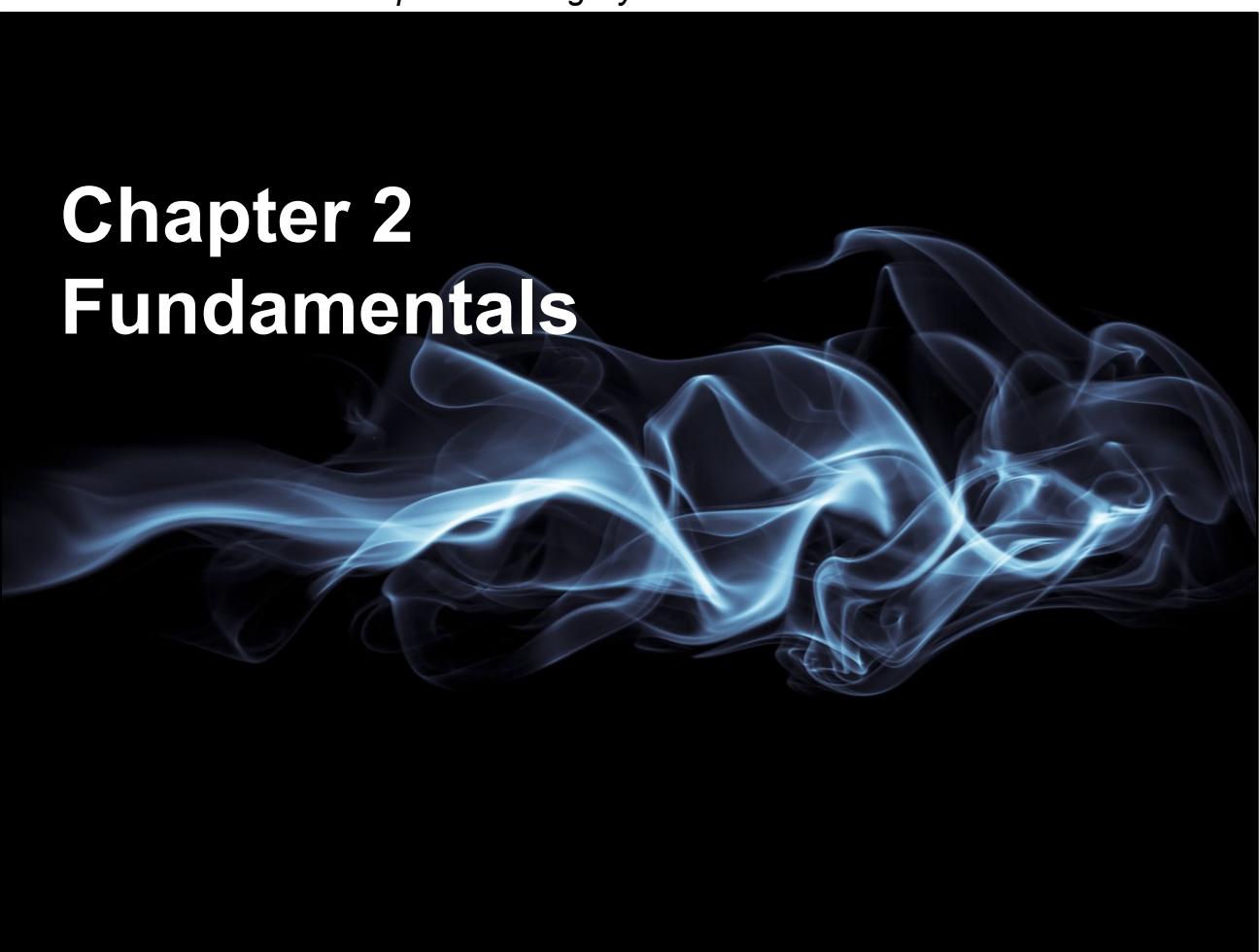
API management tools have some benefits over building APIs manually. They can be safer at providing the security features that are necessary with most APIs. They monitor API metrics (usage statistics), provide collaboration capabilities, help integrate with other systems, and require less development time to expose services. On the other hand, they come at an increased production cost and may still require considerable API development time.

# Chapter 1 Summary

- API commonly refers to RESTful services
- RESTful services expose resources on the server
  - They are accessed from a client using HTTP calls
  - Responses usually send back JSON data
  - Various HTTP methods (GET, POST, PUT, PATCH, DELETE) are used to modify the server resource
- Python provides amazing tools to simplify RESTful application development
  - Those tools include Flask and Flask plugins

# Chapter 2

# Fundamentals



# Chapter 2 Overview

## **API Fundamentals**

Creating an API

Incorporating a Database

Creating JSON Responses

# Creating a First Simple API

- In our demonstration, we'll create a RESTful app that provides celebrity data

***student\_files/data/celebrity\_100.csv***

| Name             | Pay (USD millions) | Year | Category            |
|------------------|--------------------|------|---------------------|
| Oprah Winfrey    | 225.0              | 2005 | Personalities       |
| Tiger Woods      | 87.0               | 2005 | Athletes            |
| Mel Gibson       | 185.0              | 2005 | Actors              |
| George Lucas     | 290.0              | 2005 | Directors/Producers |
| Shaquille O'Neal | 33.4               | 2005 | Athletes            |
| Steven Spielberg | 80.0               | 2005 | Directors/Producers |
| Johnny Depp      | 37.0               | 2005 | Actors              |
| Madonna          | 50.0               | 2005 | Musicians           |
| Elton John       | 44.0               | 2005 | Musicians           |
| Tom Cruise       | 31.0               | 2005 | Actors              |
| Brad Pitt        | 25.0               | 2005 | Actors              |
| Dan Brown        | 76.5               | 2005 | Authors             |
| Will Smith       | 35.0               | 2005 | Actors              |
| David Letterman  | 40.0               | 2005 | Personalities       |

The above output shows the contents of the *celebrity\_100.csv* file. This file is found in the *data* directory within the *student\_files* folder.

# Introducing Flask

- Flask is one of the top Python tools for building RESTful applications
- Since it is a third-party tool, it must be installed first
  - Use any of these (note: Flask will already be installed for you)  
(the 3.10 reference should match your Python version)

```
pip install Flask  
pip3 install Flask  
pip3.10 install Flask  
python -m pip install Flask
```



<https://flask.palletsprojects.com>

For more on Flask, visit <https://flask.palletsprojects.com/en/2.0.x/>.

Flask will be our primary production tool used. It is popular within Python, perhaps only second to Django overall. It has become an age-old (decade-old?) argument as to which environment is best for serving APIs: Flask or Django. Since Django has a considerable up front learning curve, we'll utilize Flask and be up and running in minutes.

# Defining Resources

- As part of our design process, we must decide on what our resources will be
- For an initial example, we'll use a *celebrity* object, and similarly a noun for our URI
- So, our URL will look like the following:

`http://localhost:8051/api/celebrities/tom%20cruise`

Our initial example design will be centered around our data file (celebrity\_100.csv), therefore our initial resource will be a celebrity object.

# Our First Flask Code

- Our first version instantiates the Flask server and starts it

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
app.run(host='localhost', port=8051)
```

Defines the main Flask application object

Starts the server

*student\_files/ch02\_fundamentals/01\_simple\_api.py*

There's not much to test out yet, so let's add more to it...

# Loading Some Data to Work With

- The app acquires data from a file at startup

```
from flask import Flask
from pathlib import Path

app = Flask(__name__)

data = [line.strip().split(',') for line in Path('..../data/celebrity_100.csv').open()][1:]

app.run(host='localhost', port=8051)
```

```
[['Oprah Winfrey', '225.0', '2005', 'Personalities'],
 ['Tiger Woods', '87.0', '2005', 'Athletes'],
 ['Mel Gibson', '185.0', '2005', 'Actors'],
 ['George Lucas', '290.0', '2005', 'Directors/Producers'], ...]
```

*student\_files/ch02\_fundamentals/02\_acquiring\_data.py*

In this updated version of our app, no new Flask code was added. Instead, we are merely acquiring the celebrity data from the specified file (found in the data directory of your student files). The new line of code is called a list comprehension and it builds a list containing nested lists (as shown in green at the bottom of the slide).

## Defining a Mapping

- A critical piece for working with Flask is to define which function should be called when a URI request arrives

**@app.route('path', methods=[])**

```
from flask import Flask
from pathlib import Path

app = Flask(__name__)

data = [line.strip().split(',') for line in Path('../data/celebrity_100.csv').open())[1:]

@app.route('/api/celebrities/<name>', methods=['GET'])
def do_stuff(name):
    return f'Hi there {name}!'

app.run(host='localhost', port=8051)
```

Encountering this URL (as a GET request only) will cause the do\_stuff() function to be called

*student\_files/ch02\_fundamentals/03\_creating\_a\_mapping.py*

The app.route() decorator establishes how a function is bound to a URL. It can also specify which HTTP methods are valid in order for the function to be called.

## Defining a Mapping (*continued*)

- The `app.route()` decorator gives Flask control over your functions

```
from flask import Flask  
from pathlib import Path
```

**@app.route() is a special syntax  
in Python called a **decorator****

```
data = [line.strip().split(',') for line in Path('..//data/celebrity_100.csv').open())[1:]
```

```
@app.route('/api/celebrities/<name>', methods=['GET'])  
def do_stuff(name):  
    return f'Hi there {name}!'
```

```
app.run(host='localhost', port=8051)
```

**But what is this <name> item?**

*student\_files/ch02\_fundamentals/03\_creating\_a\_mapping.py*

The full topic of decorators is beyond the scope of our course and perhaps not necessary. All that is needed to understand about decorators is that they replace the stated function with a different function. So, in this case, Flask is replacing our `do_stuff()` function with another function of their own.

# Flask Routing Styles

- Flask supports several mapping styles

```
@app.route('/api/state/<st_name>/capital/<cap_name>')
```

```
def do_stuff(st_name, cap_name):
```

```
...
```

```
@app.route('/api/email/<int:id>')
```

```
def do_stuff(id):
```

```
...
```

This type casts the value after /email/ encountered in the URL to an int type

The angle brackets, < >, allow variables to be provided in the path of the URL. Note: this is different from query string parameters which appear after the ? In the URL.

In addition to an int typecast, float may also be used.

## Testing It Out

- To test out our partial implementation, first run the Python script. If using PyCharm, right-click anywhere in the source file and select  
[Run 03\\_create\\_a\\_mapping.py](#)
- Next, in a browser navigate to  
<http://localhost:8051/api/celebrities/tom%20cruise>
- You should get a response

**Hi there tom cruise!**

# Creating a JSON Response

```
from flask import Flask, jsonify, Response
from pathlib import Path

app = Flask(__name__)

data = [line.strip().split(',') for line in Path('../data/celebrity_100.csv').open()][1:]
print('Celebrity data read.')

@app.route('/api/celebrities/<name>', methods=['GET'])
def do_stuff(name):
    results = ['value1', 'value2', 'value3']
    resp = jsonify(results=results, name=name, sample='message')
    return Response(resp.data, status=200, mimetype='application/json')

app.run(host='localhost', port=8051)
```

• Next, we'll return a JSON-based response



```
{
  "name": "tom cruise",
  "results": ["value1","value2","value3"],
  "sample": "message"
}
```

*student\_files/ch02\_fundamentals/04\_working\_with\_json.py*

To examine how to return JSON data, we provided some sample response data. Anything passed into the Flask jsonify() method will be converted into the JSON response. Values provided within jsonify() should be keyword arguments (e.g., key=value format).

# GET Single Celebrity

```

from flask import Flask, jsonify, Response
from pathlib import Path

app = Flask(__name__)

data = [line.strip().split(',') for line in Path('../data/celebrity_100.csv').open()][1:]

@app.route('/api/celebrities/<name>', methods=['GET'])
def get_one_celebrity(name):
    try:
        results = [row for row in data if name.casefold() in row[0].casefold()]
    except Exception as err:
        results = err.args

    resp = jsonify(results=results, name=name)
    return Response(resp.data, status=200, mimetype='application/json')

app.run(host='localhost', port=8051)

```

**Our final version adds logic to check celebrity name matches to the submitted value**

*student\_files/ch02\_fundamentals/05\_get\_single\_celebrity.py*

In our function, we wrote a little Python logic to iterate through our data and match the name from the URL to the name column in our data structure. We save any name matches and return those.

For those unfamiliar, the Python `casefold()` method is similar to Python's `lower()` method but works for larger character sets (containing characters from other languages).

We also renamed `do_stuff()` to `get_one_celebrity()` now.

# Flask's Request Object

- Flask provides a **request** object that will be available within your methods

Note: don't confuse the client-side **requests** module with Flask's **request** parameter

- **request** will be pre-populated with useful request data

- **remote\_addr** - (string) addr of client
- **path** - url part after the port but before the query string
- **form** - (dict) submitted form (body) params
- **args** - (dict) submitted query string params
- **headers** - (dict) client's request headers

These are only a few. There are numerous request object parameters. See the docs for more: <https://flask.palletsprojects.com/en/2.0.x/api/#incoming-request-data>.

# Flask's Request Object (1 of 2)

A client to test the Flask `request` object

```
from datetime import date
import requests
birthdate = date.today().strftime('%Y-%m-%d')
payload = {'first': 'Fred', 'middle': 'Aaron', 'pay': 3.0}
url = f'http://localhost:8051/celebrity/Savage?category=Actors&birthdate={birthdate}'
print(f'Results: {requests.post(url, data=payload).json()}')
```

```
from flask import Flask, jsonify, request, Response

app = Flask(__name__)

@app.route('/celebrity/<last_name>', methods=['POST'])
def do_stuff(last_name):
    print(f'remote_addr: {request.remote_addr}')
    print(f'path: {request.path}')
    print(f'path param: {last_name}')
    ... (continued next slide)...
```

remote\_addr: 127.0.0.1

path: /celebrity/Savage

path param: Savage

*student\_files/ch02\_fundamentals/05b\_flask\_request\_obj.py and  
05c\_testing\_flask\_request\_obj.py*

The client is shown in the top box above while our API is shown in the lower box. Our client code uses `requests` to make a request to the server. It is purposely adding additional parameters (both in the query string part of the URL and the body of the HTTP request) so that they can be extracted on the server side using Flask's `request` object.

The `requests` module (discussed earlier) is for the client-side while Flask's `request` object is used on the server side to extract out client request details.

## Flask's Request Object (2 of 2)

... (from previous slide) ...

```
print(f'form: {request.form}, sample: {request.form.get("first")}')
```

```
{'first': 'Fred', 'middle': 'Aaron', 'pay': '3.0'}, sample: Fred
```

```
print(f'args: {request.args}, sample: {request.args.get("category")}')
```

```
args: {'category': 'Actors', 'birthdate': '2022-01-19'}, sample: Actors
```

```
print(f'headers: {request.headers}, sample: {request.headers.get("User-Agent")}')
```

```
return Response(  
    jsonify({'message': 'Completed.'}).data,  
    status=200, mimetype='application/json')
```

```
headers: Host: localhost:8051  
User-Agent: python-requests/2.27.1  
Accept-Encoding: gzip, deflate  
Content-Length: 31  
sample: python-requests/2.27.1
```

```
app.run(host='localhost', port=8051)
```

*student\_files/ch02\_fundamentals/05b\_flask\_request\_obj.py and  
05c\_testing\_flask\_request\_obj.py*

This is a continuation of the code presented on the previous slide.

## Your Turn! - Task 2-1

- Create a RESTful app to retrieve invoice data given an invoice order number
  - Valid invoice order numbers are: **536365 - 581587**
  - *Implement only the HTTP GET method*
- Use the Flask starter code provided in ***ch02\_fundamentals/task2\_1\_starter.py***

InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country  
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,12/1/2010 8:26,2.55,17850,United Kingdom  
536365,71053,WHITE METAL LANTERN,6,12/1/2010 8:26,3.39,17850,United Kingdom  
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,12/1/2010 8:26,2.75,17850,United Kingdom  
536365,84029G,KNITTED UNION FLAG HOT WATER BOTTLE,6,12/1/2010 8:26,3.39,17850,United Kingdom  
536365,84029E,RED WOOLLY HOTTIE WHITE HEART.,6,12/1/2010 8:26,3.39,17850,United Kingdom  
536365,22752,SET 7 BABUSHKA NESTING BOXES,2,12/1/2010 8:26,7.65,17850,United Kingdom  
536365,21730,GLASS STAR FROSTED T-LIGHT HOLDER,6  
536366,22633,HAND WARMER UNION JACK,6,12/1/2010  
536366,22632,HAND WARMER RED POLKA DOT,6,12/1/2010  
536367,84879,ASSORTED COLOUR BIRD ORNAMENT,32,12  
F7/7/7 007/E DODDYS PLAVHOUSE BEDROOM / 12/1/2010 8:27 0 1 170/7 United Kingdom

For now, test it out in a browser using invoice 536365.  
Also, test it out using invoice 1.

# Consuming APIs

- The Python requests module could be used to consume API data

```
import json
import requests

base_url = 'http://localhost:8051'
path = '/api/celebrities/'
default = 'Kevin'

celeb_name = input('Enter celebrity to find info about (def. Kevin): ')
if not celeb_name:
    celeb_name = default

results = requests.get(f'{base_url}{path}{celeb_name}').json()

http://localhost:8051/api/celebrities/Kevin

print(json.dumps(results, indent=4))
```

*student\_files/ch02\_fundamentals/06\_consumng\_apis.py*

The variable 'results' will be a Python dictionary. It would look (in part) like this:

```
{
    "name": "Kevin",
    "results": [
        [
            "Kevin Garnett",
            "29.0",
            "2008",
            "Athletes"
        ],
        [
            "Kevin Spacey",
            "16.0",
            "2014",
            "Television actors"
        ],
        [
            "Kevin Hart",
            "87.5",
            "2016",
            "Comedians"
        ],
        [
            "Kevin Hart",
            "39.0",
            "2020",
            "Comedians"
        ]
    ]
}
```

# Using *requests* to Check HTTP Status

```
import requests

r = requests.get('http://httpbin.org/status/200')

if r.ok:
    print('Request was successful.')

if r.status_code == 200:
    print('Request was successful.')

if r:
    print('Requests returned a non-400 or 500 error code.')

r = requests.get('http://httpbin.org/status/400')
try:
    r.raise_for_status()
except requests.exceptions.HTTPError as err:
    print(err)
```

*student\_files/ch02\_fundamentals/07\_checking\_status.py*

The third conditional above will return true for any non-400 or 500 error status codes. The bottom example shows how the request could be checked for any kind of errors returned in the response and then raise an exception because of it.

## Your Turn! - Task 2-2

- Create a *Python client* to access the invoicing API created in task 2-1

`http://localhost:8051/api/invoices/536365`



- Work from the provided `task2_2_starter.py` file
- Ensure your Flask server from task 2-1 is running before testing your answer
  - Access invoice `536365` and display its results

# Python Classes

- Classes allow for the grouping of attributes and methods

```

from pathlib import Path
data (below) is passed into the __init__
class Celebrity:
    def __init__(self, name, pay, year, category):
        self.name = name
        self.pay = pay
        self.year = year
        self.category = category

    def __str__(self):
        return f'{self.year:<6}{self.name} ({self.category}) ${self.pay:>} million'

f = Path('../data/celebrity_100.csv').open(encoding='utf-8')
f.readline() # reads the header
data = f.readline().strip().split(',') ['Oprah Winfrey', '225.0', '2005', 'Personalities']

celeb = Celebrity(*data)
print(celeb)
2005 Oprah Winfrey (Personalities) $225.0 million

```

*student\_files/ch02\_fundamentals/08\_classes.py*

Classes are commonly used within Python. The `self` parameter is required for all class methods (with a couple of exceptions not important here). The `__init__()` method will be called when an object instance is created of our class. Within the `__init__()` we attached the values passed in to our new object instance (a celebrity object called `celeb`).

# Introducing Flask-RESTX



- **Flask-RESTX** is an addon to Flask to enable easier class-based creation of Resources

- As it is a third-party tool, it must be installed
    - Try any of these (note: you will not need to install this if you performed Task1-2 earlier)

```
pip install flask-restx  
pip3 install flask-restx  
pip3.10 install flask-restx  
python -m pip install flask-restx
```

- Flask-RESTX is a compatible fork of Flask-RESTplus
  - Easy to use
  - Works in Python 3.4+

**Flask-RESTX Documentation**

<https://flask-restx.readthedocs.io/en/latest/>

Flask-RESTX is backward compatible with Flask-Restful and Flask-RestPlus but provides several additional features. Since it is the most currently maintained, this is the plugin we will use.

# Using Flask-RESTX

```

from flask import Flask
from flask_restx import Resource, Api

app = Flask(__name__)
api = Api(app)

data = [line.strip().split(',') for line in Path('../data/celebrity_100.csv').open(encoding='utf-8')][1:]

@api.route('/api/celebrities/<name>')
class Celebrity(Resource):
    def get(self, name):
        try:
            results = [row for row in data if name.casifold() in row[0].casifold()]
        except Exception as err:
            results = err.args

        return {'results': results, 'name': name}

app.run(host='localhost', port=8051)

```

Most Flask plugins are implemented by passing the `app` object into the plugin's main object. Flask-RESTX's main object is called `Api()`.

Extend a class from `Resource`

`student_files/ch02_fundamentals/09_flask_restx.py`

You can either try `http://localhost:8051/api/celebrities/Kevin` in your browser directly or you can run the example client code discussed earlier (`06_consumming_apis.py`).

This example now uses flask-RESTX which makes it easy to build classes that can contain the required methods to build a Resource. Classes should inherit from `Resource`.

# Adding Other RESTful Operations

A second class is added for the plural URL conditions (GET all, POST)

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

app = Flask(__name__)
api = Api(app, prefix='/api')

class Celebrities(Resource):
    def get(self):
        return {'celebrities': data}

    def post(self):
        print('post', api.payload)
        return {'action': 'post'}
```

Our class now supports the other RESTful (singular) methods

```
class Celebrity(Resource):
    def get(self, name):
        # unchanged from previous

    def delete(self, name):
        return {'action': 'delete'}

    def put(self, name):
        return {'action': 'put'}
```

*/api/celebrities/Kevin*

How we'll define our mappings now

```
api.add_resource(Celebrity, '/celebrities/<name>')
api.add_resource(Celebrities, '/celebrities/')

app.run(host='localhost', port=8051)
```

*student\_files/ch02\_fundamentals/10\_adding\_other\_operations.py*

This is a nice simplified version of our API represented across two classes. Each class is used for a different endpoint URI.

We defined our URL mappings now without the use of a decorator. This can be an easier alternative to placing decorators above the class names.

# A Test Client

```
base_url = 'http://localhost:8051'
path = '/api/celebrities/'
celeb_name = 'Kevin'
```

```
results = requests.get(f'{base_url}{path}{celeb_name}')
print(results.text)
```

This test client uses Python's requests framework to make requests to our REST API server

{"results": [{"name": "Kevin Garnett", "category": "Actors", "pay": 29.0, "year": 2022}]}  
{"action": "get"}

```
r = requests.post(f'{base_url}{path}',
                  data={'name': celeb_name, 'category': 'Actors',
                        'pay': 3.0, 'year': 2022})
print(results.text)
```

{"action": "post"}

```
results = requests.put(f'{base_url}{path}{celeb_name}')
print(results.text)
```

{"action": "put"}

```
print('deleting:')
results = requests.delete(f'{base_url}{path}{celeb_name}')
print(results.text)
```

{"action": "delete"}

```
print('get all:')
results = requests.get(f'{base_url}{path}')
print(results.text)
```

(displays all records)

*student\_files/ch02\_fundamentals/11\_testing\_other\_operations.py*

The client here can make requests to our previously shown Flask server containing the Celebrity and Celebrities classes.

## Your Turn! - Task 2-3

- Incorporate Flask-RESTX into our previous invoice application (Task 2-1)
  - Keep the URIs the same as Task 2-1
- Create two classes: *Invoice* and *Invoices*
- Add a *get()* and *post()* method to Invoices and a *get()*, *delete()*, and *put()* method to Invoice
- Place temporary return values in each of the methods
- Use the provided test client to test your solution  
*(task2\_3\_client.py)*

Work from your finished *task2\_1\_starter.py* file or the provided *task2\_3\_starter.py* file

# Postman and Testing APIs

- Postman is a popular client for creating and testing APIs

- Runs on Windows, Mac, Linux
- Postman has several tiers for usage including a Free plan for small teams

**Homepage: <https://www.postman.com/>**

- Can be used from a web-based interface, or

**Web-based:  
<https://web.postman.co/>**

**For localhost, the desktop version must be used**

- As a desktop app installed from

**<https://www.postman.com/downloads/>**

**Either the web or desktop versions require creating an account (free)**

The screenshot shows the Postman website at <https://www.postman.com/downloads/>. It highlights the 'Download Postman' section, which includes a note about using the desktop version for localhost. Below this, it shows a 'Web-based' option at <https://web.postman.co/>. A large blue box highlights the download link: <https://www.postman.com/downloads/>. To the right, a screenshot of the Postman desktop application interface is shown, displaying a collection of API requests for the Twitter API v2.

Postman is widely used and is offered at several levels depending on needs and team size. For many, the free plan will suffice, but for large-scale collaborative needs, higher-level plans may be necessary.

To test using localhost as the domain of the URL, the desktop app version must be used.

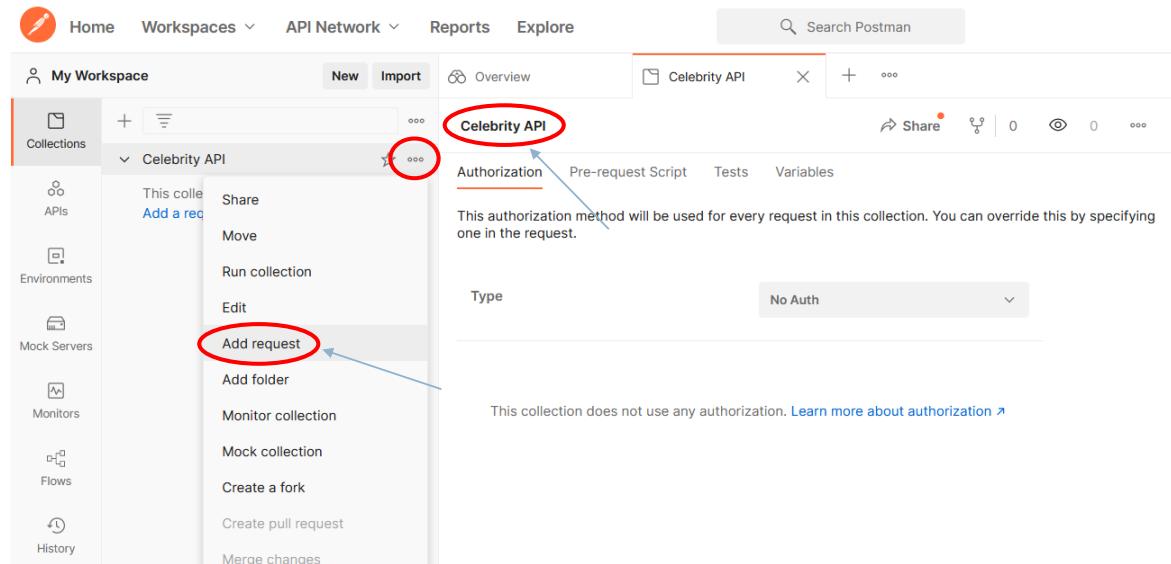
# Using Postman: Creating Collections

The screenshot shows the Postman application interface. At the top, there is a navigation bar with links for Home, Workspaces, API Network, Reports, Explore, and a search bar labeled 'Search Postman'. On the far right of the top bar are icons for workspace settings, invite, notifications, and help.

The main area is titled 'My Workspace'. It features a sidebar on the left with icons for Home, Workspaces, API Network, Reports, Explore, and a 'Collections' icon which is circled in red. Below the sidebar, there is a list of APIs under 'APIs' and a section titled 'Create a collection for your requests' with a 'Create collection' button. The central area displays a collection named 'Media Authentication API' containing several requests like 'POST Authentication', 'POST Access Token', etc. To the right of the collection list is a summary section with fields for 'Add a brief summary about this workspace' and 'In this workspace' which lists 'Requests (0)', 'Collections (0)', 'APIs (0)', 'Environments (0)', 'Mock Servers (0)', and 'Monitors (0)'. At the bottom of the workspace area is an 'Activity' section.

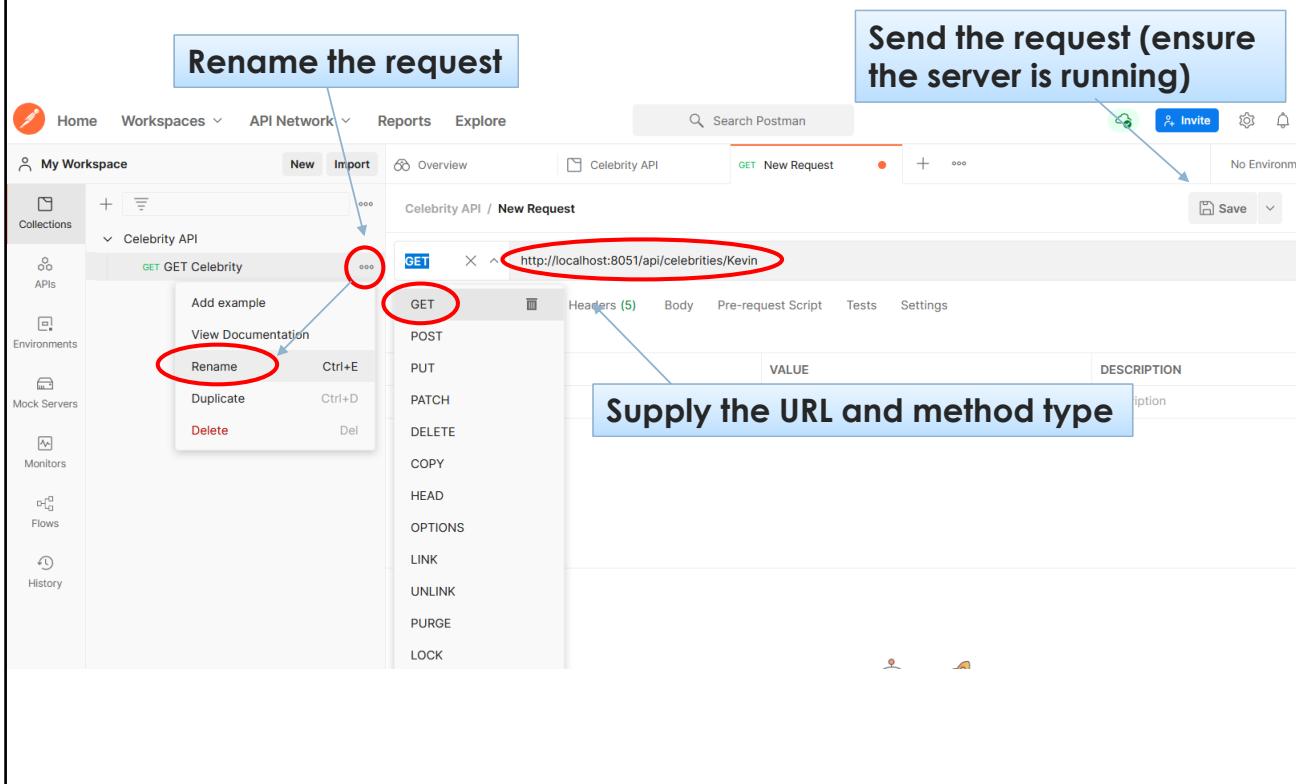
On the screen shown above, the browser-based version loaded the **web.postman.co** url and signed in. Next, a collection was created. Collections are groupings of URLs (typically related to an API set).

# Using Postman: Adding Requests



Once a collection is created, you can give the overall API a name and define API requests.

# Using Postman: Sending Requests



By supplying the URL and method type for the request, it can easily be sent or re-sent anytime.

# Using Postman: Setting Params, Etc.

**Optionally set params (query string)**

The screenshot shows the Postman interface for a GET request to `http://localhost:8051/api/celebrities/Kevin?foo=bar`. The 'Params' tab is selected, and a parameter `foo` is set to `bar`. Other tabs include Authorization, Headers (5), Body, Pre-request Script, Tests, and Settings.

| KEY | VALUE |
|-----|-------|
| foo | bar   |
| Key | Value |

**Optionally set headers**

The screenshot shows the Postman interface for a GET request to `http://localhost:8051/api/celebrities/Kevin?foo=bar`. The 'Headers' tab is selected, showing several auto-generated headers like `Postman-Token`, `Host`, `User-Agent`, etc. A blue box labeled 'Send the request' points to the 'Send' button.

| KEY             | VALUE                             |
|-----------------|-----------------------------------|
| Postman-Token   | <calculated when request is sent> |
| Host            | <calculated when request is sent> |
| User-Agent      | PostmanRuntime/7.28.4             |
| Accept          | /*                                |
| Accept-Encoding | gzip, deflate, br                 |
| Connection      | keep-alive                        |

**View results**

The screenshot shows the Postman interface displaying the JSON response under the 'Test Results' tab. The response contains a list of celebrities with their names, birth years, death years, and professions.

```
{
  "results": [
    ["Kevin Garnett", "1969", "2009", "Athletes"], ["Kevin Garnett", "98.0", "2009", "Athletes"], ["Kevin Hart", "84.0", "2013", "Comedians"], ["Kevin Durant", "92.0", "2014", "Athletes"], ["Kevin Spacey", "66.0", "2014", "Television actors"], ["Kevin Durant", "54.0", "2015", "Athletes"], ["Kevin Hart", "28.5", "2015", "Comedians"], ["Kevin Hart", "87.5", "2016", "Comedians"], ["Kevin Durant", "56.0", "2015", "Athletes"], ["Kevin Durant", "60.6", "2017", "Athletes"], ["Kevin Hart", "32.5", "2017", "Comedians"], ["Kevin Hart", "57.0", "2018", "Comedians"], ["Kevin Durant", "53.7", "2018", "Athletes"], ["Kevin Durant", "65.4", "2019", "Athletes"], ["Kevin Hart", "59.0", "2019", "Comedians"], ["Kevin Durant", "63.9", "2020", "Athletes"], ["Kevin Hart", "39.0", "2020", "Comedians"]]
  ],
  "name": "Kevin"
}
```

Postman provides a nice degree of customization of requests including the ability to set query string params, headers, and body parameters.

# Your Turn! - Task 2-4 (optional)

- Use Postman to verify our Invoice application
- Test each of the endpoints and methods created in the previous exercise

The screenshot shows the Postman application interface. On the left, the sidebar lists collections, APIs, environments, mock servers, monitors, flows, and history. The main workspace shows a collection named "Invoice API / GET Invoice" containing four requests: "GET GET Invoices", "POST POST Invoices", "GET GET Invoice", and "PUT PUT Invoice". The "GET GET Invoice" request is currently selected, showing its details. The URL is set to "http://localhost:8051/api/invoices/536365". The "Body" tab of the request details panel is active, displaying the following JSON response:

```

{
  "results": [
    {
      "id": "85123A",
      "name": "WHITE HANGING HEART T-LIGHT HOLDER",
      "size": "6",
      "date": "12/1/2010 8:26",
      "price": "2.55",
      "country": "United Kingdom"
    },
    {
      "id": "71053",
      "name": "WHITE METAL LANTERN",
      "size": "6",
      "date": "12/1/2010 8:26",
      "price": "3.39",
      "country": "United Kingdom"
    },
    {
      "id": "84406B",
      "name": "CREAM CUPID HEARTS COAT HANGER",
      "size": "8",
      "date": "12/1/2010 8:26",
      "price": "2.75",
      "country": "United Kingdom"
    },
    {
      "id": "840296",
      "name": "KNITTED UNION FLAG HOT WATER BOTTLE",
      "size": "6",
      "date": "12/1/2010 8:26",
      "price": "3.39",
      "country": "United Kingdom"
    },
    {
      "id": "84029E",
      "name": "RED WOOLLY HOTTIE WHITE HEART",
      "size": "6",
      "date": "12/1/2010 8:26",
      "price": "3.39",
      "country": "United Kingdom"
    },
    {
      "id": "22752",
      "name": "SET 7 BABUSHKA NESTING BOXES",
      "size": "2",
      "date": "12/1/2010 8:26",
      "price": "7.65",
      "country": "United Kingdom"
    },
    {
      "id": "21730",
      "name": "GLASS STAR FROSTED T-LIGHT HOLDER",
      "size": "6",
      "date": "12/1/2010 8:26",
      "price": "4.25",
      "country": "United Kingdom"
    }
  ]
}

```

# Bringing In the Database

Flask  SQLAlchemy



- A commonly used Python tool for interacting with a database is called **SQLAlchemy**  
<https://flask-sqlalchemy.palletsprojects.com/en/2.x/>
- SQLAlchemy performs *automatic Python object-to-relational database mapping* (ORM)
- SQLAlchemy is a third-party tool and must be installed  
`pip install sqlalchemy`
- **Flask-SQLAlchemy** is a plugin that integrates the two frameworks
  - Flask plugins must be installed normally
  - *Our setup already includes both of these*

SQLAlchemy provides a means to integrate the Python world with the database. It not only converts relations to objects (and vice-versa), but it takes care of the database SQL statements as well.

Flask supports a plugin called Flask-SQLAlchemy that can integrate the two frameworks, Flask and SQLAlchemy, making integration with most relational databases an easy task.

# Configuring SQLAlchemy with Flask

- Three steps to get SQLAlchemy up and running
  - Configure the Flask-SQLAlchemy plugin (shown below)
  - Define models to interact with the database
  - Invoke model methods to perform database operations

- Step 1.**  
Configure the plugin by defining how to connect to the database

```
from flask import Flask, request
from flask_restx import Resource, Api
from flask_sqlalchemy import SQLAlchemy

student_files_dir = Path(__file__).parents[1]
db_file = student_files_dir / 'data/course_data.db'

app = Flask(__name__)
api = Api(app, prefix='/api')

app.config['SQLALCHEMY_DATABASE_URI'] =
    'sqlite:///{} + str(db_file)

db = SQLAlchemy(app)
```

This is the location of our database

*student\_files/ch02\_fundamentals/12\_flask\_sqlalchemy\_post.py*

Configuring the Flask-SQLAlchemy plugin is easy. We added a parameter called `SQLALCHEMY_DATABASE_URI` to the Flask app's config object. When the Flask-SQLAlchemy plugin starts up, it will look within the Flask app's config object for this value. For other databases, other properties may be required, such as properties for a username and password to authenticate with the database.

As is done with most plugins, the `app` object is passed into the `SQLAlchemy` object.

Not shown on the slide but found in the source code is the `track modification config` parameter. The track modification config parameter is a feature that is being removed from the current version of the plugin. However, without this statement turning it off, it will cause a warning message to appear when the server starts. So, we merely turn it off.

# Defining and Using a Model

- Step 2.  
Create a model class to map to the database

```
app = Flask(__name__)
api = Api(app, prefix='/api')

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///{} + str(db_file)
db = SQLAlchemy(app)

class CelebrityModel(db.Model):
    __tablename__ = 'celebrity'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    pay = db.Column(db.Float)
    year = db.Column(db.String(15))
    category = db.Column(db.String(50))

    def __init__(self, name, pay, year, category):
        self.name = name
        self.pay = pay
        self.year = year
        self.category = category

    def __str__(self):
        return f'{self.year} {self.name} {self.pay} {self.category}'
```

**The model inherits from db.Model**

**Define the fields and types to map to the table**

**The \_\_init\_\_ and \_\_str\_\_ are optional but can be helpful**

*student\_files/ch02\_fundamentals/12\_flask\_sqlalchemy\_post.py*

A model class should inherit from db.Model (db was created from the Flask-SQLAlchemy plugin object). Then (at the class level!) add field definitions to the class that correspond to the fields that will map to the database. Optional \_\_init\_\_() and \_\_str\_\_() methods are provided for convenience for instantiation and debugging or logging purposes.

# API POST with Flask-SQLAlchemy

- Step 3.  
We'll modify our Flask-RESTX class to interact with a database

```
class Celebrities(Resource):
    def get(self):
        return {'celebrities': ""}

    def post(self):
        name = request.form.get('name')
        year = request.form.get('year')
        category = request.form.get('category')
        pay = float(request.form.get('pay'))

        new_celeb = CelebrityModel(name, pay, year, category)
        db.session.add(new_celeb)
        db.session.commit()

        return {'id': new_celeb.id, 'name': name, 'year': year,
                'pay': pay, 'category': category}
```

**Our POST method extracts the new parameters from the HTTP request body**

**Instantiate a model, perform an INSERT**

**Commonly, the inserted object is returned**

Test the POST operation either using Postman or by running  
13\_testing\_flask\_sqlalchemy\_post.py

Our POST method of the Celebrities class has been modified. It now extracts the HTTP request (body) parameters using Flask's provided (injected) request object. There are other ways to obtain the HTTP request data as well. This data is provided to the model class and then it is inserted into the database (via an SQL INSERT operation) using the db object session's add() method. We commonly return to the client a JSON-based form of the object that was inserted. Here, we returned that object by manually creating it.

Shortly, we'll use a handy tool, Flask-Marshmallow, to serialize the object to JSON for us.

## About the Database

- The database used in the course is a SQLITE database file found in the *data* directory (called `course_data.db`)
- Since the DB tables will now have an **id** (primary key) column, we'll modify our solution slightly to work with the id column for GET, PUT, and DELETE

**Note:** When interacting with the database, it can be reset anytime by simply running the `restore_db.py` file in the `student_files/data` directory.

This resets the tables back to their original datasets.

# Our New Celebrity Resource

```
class Celebrities(Resource):
    def get(self):
        return {'celebrities': []}

    def post(self):
        name = request.form.get('name')
        year = request.form.get('year')
        category = request.form.get('category')
        pay = float(request.form.get('pay'))

        new_celeb =
            CelebrityModel(name, pay, year, category)
        db.session.add(new_celeb)
        db.session.commit()

        return {'id': new_celeb.id, 'name': name,
               'year': year, 'pay': pay, 'category': category}

api.add_resource(Celebrities, '/celebrities/')
```

```
class Celebrity(Resource):
    def get(self, id):
        return {'action': 'get'}

    def delete(self, id):
        return {'action': 'delete'}

    def put(self, id):
        return {'action': 'put'}
```

api.add\_resource(Celebrity, '/celebrities/<id>')

The other 4 methods will be implemented shortly

*student\_files/ch02\_fundamentals/12\_flask\_sqlalchemy\_post.py*

Shown are our two Resource classes. Four of the methods are not implemented yet. The get(), put(), and delete() methods have been adjusted slightly since we are now using a database. The database table has a primary key that can now be used as a path parameter when making requests.

## Your Turn! - Task 2-5

- Add [Flask-SQLAlchemy](#) to the current Invoice application
- Complete the API's POST method found in the Invoices class
- Either work from your completed Task 2-3 or from the provided `task2_5_starter.py` file
- Test it by running the provided `task2_5_client.py` file or by running it within Postman

# Flask-Marshmallow



- What is Flask-marshmallow?
  - Flask-marshmallow is a helpful tool for converting objects from Python to JSON or JSON to Python
  - Ideal for returning a Flask database object as JSON from an API

Marshmallow homepage: <https://marshmallow.readthedocs.io/en/stable/>

Flask-Marshmallow (plugin) home: <https://flask-marshmallow.readthedocs.io/en/latest/>

- To use Marshmallow:
  1. [Configure](#) the plugin within Flask
  2. [Define and instantiate a "schema"](#) (the fields to be serialized)
  3. [Use the schema](#) to return from API methods (as needed)

Marshmallow is a standalone tool within Python. Marshmallow by itself is useful in converting Python objects to dictionaries. While numerous tools can do this, some of them have problems if the objects do not have underlying dictionaries. In other cases, the utilities for doing this are framework-specific (like Flask's `jsonify()` ).

Flask-Marshmallow simply takes the easy-to-use marshmallow framework and brings it into Flask (as has been done with other plugins like SQLAlchemy and RESTX).

Here, we make use of it to automatically convert the database SQLAlchemy objects into JSON directly.

# Installing Flask-Marshmallow

- **Flask-marshmallow** is third-party module and must be installed using the appropriate statement

```
pip install flask-marshmallow  
pip3 install flask-marshmallow  
pip3.10 install flask-marshmallow  
python -m pip install flask-marshmallow
```

- When using SQLAlchemy objects with Marshmallow, another plugin, **Marshmallow-SQLAlchemy**, should also be installed
- *Neither of these need to be installed for our environment*

```
pip install marshmallow-sqlalchemy  
pip3 install marshmallow-sqlalchemy  
pip3.10 install marshmallow-sqlalchemy  
python -m pip install marshmallow-sqlalchemy
```

When using both SQLAlchemy and Marshmallow (specifically for converting SQLAlchemy objects) it is best to install two items: flask-marshmallow and marshmallow-sqlalchemy.

Don't be confused with these tools. All they do is provide an automatic conversion of Python objects to JSON format.

# Integrating Flask-Marshmallow Is Easy

```
from flask import Flask, request
from flask_marshmallow import Marshmallow

app = Flask(__name__)
ma = Marshmallow(app)

class CelebrityModel(db.Model):
    ...
class Celebrities(Resource):
    ...
class Celebrity(Resource):
    ...

class CelebritySchema(ma.Schema):
    class Meta:
        fields = ('id', 'name', 'year', 'pay', 'category')

celebrity_schema = CelebritySchema()
celebrities_schema = CelebritySchema(many=True)
app.run(host='localhost', port=8051)
```

**Step 1. Configure the plugin**

**Step 2. Define and instantiate schema**

`student_files/ch02_fundamentals/14_flask_sqlalchemy_marshmallow.py`

Configuring Flask-Marshmallow is only two lines: an import and passing the Flask app object into the Marshmallow main object. Next, define a class that identifies which fields will be exposed (serialized). Instantiate these so that the instances can be used in your API methods (next slide).

# POST Again with Flask-Marshmallow

```
class Celebrities(Resource):
    def get(self):
        return {'celebrities': ''}
```

`celebrity_schema.jsonify(obj)` - returns **JSON**

`celebrity_schema.jsonify(obj)` - returns a **dict**

```
def post(self):
    name = request.form.get('name')
    year = request.form.get('year')
    category = request.form.get('category')
    pay = float(request.form.get('pay'))
```

```
new_celeb = CelebrityModel(name, pay, year, category)
db.session.add(new_celeb)
db.session.commit()
```

```
return celebrity_schema.jsonify(new_celeb)
```

Step 3. The Marshmallow schema definition will now convert your (SQLAlchemy) objects to JSON

Test the POST operation again (this time with Marshmallow included) either using Postman or by running `15_testing_flask_sqlalchemy_marshmallow.py`

`student_files/ch02_fundamentals/14_flask_sqlalchemy_marshmallow.py`

Hey, couldn't Flask's jsonify() do that? Well, Flask's jsonify() can't easily convert Python objects to JSON.

Can't the Python json module help? Sure, but there's a bit of manual work involved with using it and a bunch of work if the object is not serializable.

Marshmallow tries to provide a convenient way to handle object-to-dict or object-to-json conversions.

# GET and GET-all with Flask-SQLAlchemy

```

class Celebrities(Resource):
    def get(self):
        all_celebs = CelebrityModel.query.all()
        return {'results': celebrities_schema.dump(all_celebs)}

    def post(self):
        (unchanged from previous)

class Celebrity(Resource):
    def get(self, name):
        celeb = CelebrityModel.query.get(id)
        return celebrity_schema.jsonify(celeb)

    def delete(self, name):
        return {'action': 'delete'}

    def put(self, name):
        return {'action': 'put'}

```

Performs equivalent to a  
SELECT \* FROM celebrity

Query the database and find  
the record that matches the  
specified id primary key

Test the two GET operations either  
using Postman or by running  
`17_testing_flask_sqlalchemy_get.py`

`student_files/ch02_fundamentals/16_flask_sqlalchemy_get.py`

Now that the pieces are in place, the remaining methods should come along easily. The two get methods are shown above. The first is an API GET handler that returns matches to a celebrity name. The second returns all celebrity names.

`schema.dump()` - converts the object to a native Python type. Appropriate if you are wrapping additional code around the object (as we do in the case of the Celebrities `get()` above).

`schema.jsonify()` - simply turns the specified object into a JSON response. Useful when you are returning the object only (as we did in the case of the `post()` method on the previous slide) or the `Celebrity get()` method on this slide.

# API PUT / DELETE with Flask-SQLAlchemy

```
class Celebrity(Resource):
    def get(self, id):
        (as shown previously)

    def delete(self, id):
        celeb = CelebrityModel.query.get(id)
        db.session.delete(celeb)
        db.session.commit()
        return celebrity_schema.jsonify(celeb)

    def put(self, id):
        year = request.form.get('year')
        category = request.form.get('category')
        pay = float(request.form.get('pay'))

        celeb = CelebrityModel.query.get(id)
        celeb.year = year
        celeb.category = category
        celeb.pay = pay

        db.session.commit()
        return celebrity_schema.jsonify(celeb)
```

Using the object id, lookup the object in the database, then use the object to perform a delete

Get the submitted data, instantiate a model, make any changes (e.g., celeb.year = year)

Test the two GET operations either using Postman or by running 19\_testing\_flask\_sqlalchemy\_put.py

*student\_files/ch02\_fundamentals/18\_flask\_sqlalchemy\_put.py*

The final methods are implemented here. put() and delete() are easily implemented now.

## Implementing PATCH

- Our PUT operation performs a full update (as much as we are allowed) on the Celebrity object
- But what if we only wanted to update the category and not the pay or year attributes?
- Use PATCH for partial updates

```
def patch(self, id):
    celeb = CelebrityModel.query.get(id)

    if 'year' in request.form:
        celeb.year = request.form.get('year')
    if 'category' in request.form:
        celeb.category = request.form.get('category')
    if 'pay' in request.form:
        celeb.pay = float(request.form.get('pay'))

    db.session.commit()

    return celebrity_schema.jsonify(celeb)
```

*student\_files/ch02\_fundamentals/20\_flask\_sqlalchemy\_patch.py*

The PATCH method is best suited for situations where only a part of an object will be updated. The PUT method can also still be implemented but is best when a full object update will occur.

## Your Turn! - Task 2-6

- Add **Flask-Marshmallow** to the current Invoice application
- Complete the remaining API methods: `get()` found in the `Invoices` class, and `get()`, `put()`, and `delete()` found in the `Invoice` class
- Either work from your completed Task 2-5 file or work from the provided `task2_6_starter.py` file
- Test it by running the provided `task2_6_client.py` file or by running it within Postman

# Error Handling and Common HTTP Statuses

- Best practices dictate that any error encountered on the server should return an appropriate 400 (usually) or 500 HTTP status code

| Status Code | Description                                                                          |
|-------------|--------------------------------------------------------------------------------------|
| 400         | <b>Bad Request</b> (Usually no URL mapping)                                          |
| 401         | <b>Unauthorized</b> (Usually when client is not authenticated)                       |
| 403         | <b>Forbidden</b> (Usually when client is not authorized but is authenticated)        |
| 404         | <b>Not found</b> (Good when resource id isn't found)                                 |
| 405         | <b>Method not allowed</b> (e.g., A POST occurred on a GET only method)               |
| 429         | <b>Too many requests</b> (User sent too many requests in a given period of time)     |
| 500         | <b>Internal Server Error</b> (Usually a code execution error occurred on the server) |

There are many more HTTP status codes.

A good reference is: <https://restfulapi.net/http-status-codes/>.

APIs should return an appropriate status code based on the condition of the error. Don't return 200 when the application has a logic error (even if the request was processed properly). Send back an appropriate 4xx code.

# Non-Error Status Values

- There are other return status values that can be used to indicate non-error conditions

| Status Code | Description                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| 200         | <a href="#">Ok</a> (The majority of response status codes are 200)                                                   |
| 201         | <a href="#">Created</a> (The server created a new resource)                                                          |
| 204         | <a href="#">No Content</a> (The request processed successfully, but the server didn't have anything to respond with) |
| 300         | <a href="#">Multiple Choices</a> (The request has multiple possible responses)                                       |
| 304         | <a href="#">Not modified</a> (Server code is not changed, client's cached version is okay to use)                    |

Even though most successful responses will return 200 status, sometimes other values might make sense. For example, a 204 status code could be sent back in situations where a request was processed successfully but the server didn't have any message to return.

# Handling Errors with Flask-RESTX

- The Flask-RESTX plugin provides an `abort()` method that will send back an HTTPException

```
def handle_no_celebrity_exists_error(message: str = f'The celebrity doesn\'t exist.',  
                                     error_code: int = 404):  
    abort(error_code, message)  
  
class Celebrity(Resource):  
    def get(self, id):  
        celeb = CelebrityModel.query.get(id)  
        if not celeb:  
            handle_no_celebrity_exists_error()  
  
        return celebrity_schema.jsonify(celeb)
```

The custom method internally calls Flask-RESTX's `abort()` method

`student_files/ch02_fundamentals/22_flask_sqlalchemy_errors.py`

Here, a custom method was created that will invoke the Flask-RESTX `abort()` method for us. Now this method can be applied to `get()`, `put()`, `patch()`, and `delete()` methods and a custom JSON message is then sent back to the client.

Notice we don't send back a 200 when an application error (like no celebrity found) occurs.

## Chapter 2 Summary

- Python provides tools to facilitate building all aspects of APIs
- [Flask](#) is only one of many options for serving content
- The Flask environment provides numerous options for improving solutions via plugins, including:
  - [Flask-RESTX](#) (and earlier variations -ful/-plus) simplify the amount of coding required
  - [Flask-SQLAlchemy](#) provides database integration and ORM
  - [Flask-Marshmallow](#) provides easy object return state

# Chapter 3

# Advanced Concepts



# Chapter 3 Overview

## **Advanced Concepts**

Improving Performance

Caching

Versioning

Rate Limiting

Contract-First Development

HATEOAS

# Improving Performance

- There are several ways in which application performance can be improved
  - [Pagination](#) - Limits the amount of data returned for large responses
  - [Caching](#) - Lets the client re-use data saved from previous responses
  - [Rate Limits](#) - Limit a client's number of requests over time
  - [Versioning](#) - Less for performance than to ensure continuity for clients

Let's explore these techniques over the remainder of this chapter...

# Pagination

```
class Celebrities(Resource):
    page_size = 10
```

Pages are 10 records

```
def get(self):
```

`page = request.args.get('page') or '1'`

`paged_celebs = CelebrityModel.query.paginate(int(page), self.page_size)`

`return {'results': celebrities_schema.dump(paged_celebs.items)}`

A page query parameter  
(e.g., `page=2`) or def. 1

```
import requests
```

`base_url = 'http://localhost:8051'`

`path = '/api/celebrities/'`

```
for n in range(1, 4):
```

`print(f'\ngetting page ({n}):')`

`results = requests.get(f'{base_url}{path}', params={'page': n})`

`print(f'Status: {results.status_code}')`

`print(results.text)`

Flask SQLAlchemy provides a `paginate()` method that takes a **page number** and **page size**. It returns a Paginate object that has an **items** attribute

Our test client retrieves 3 pages of 10 records per page

`student_files/ch03_adv_concepts/01_pagination.py`

In the code shown above, we see our Celebrities resource. We've defined our own `page_size` variable. This defines how many records will be returned. It is arbitrarily defined but can be easily changed now.

Also, the `page_size` variable could be made into something the client could manipulate and send to our server if we desired.

We used the Flask `request` object to obtain the submitted page parameter from the URL query string.

# Caching

- Caching improves API performance by minimizing the amount of client requests made to the server
    - Caching only applies to GET requests
    - Caching can be implemented by setting just a couple of HTTP response headers
    - Use the Cache-Control header to define if and for how long response data can be saved
      - Cache-Control: max-age=600** *(use browser cache if request is within 10 minutes of original)*
      - Cache-Control: no-cache** *(don't cache the response)*
- With this header, browser-based clients will look toward their own cache implementation before making a request

Caching within browsers is automatic, however, non-browser clients may require logic to process the header. Caching only works in one direction. In other words, if the request indicates it is not to be cached with a Cache-Control header, it will only be obeyed in one direction (client to server). The response would have to indicate with its own header whether to cache the response info.

It's worth noting that leaving off the Cache-Control header doesn't necessarily disable caching. Browsers can attempt to employ caching on their own for certain types of content. The no-cache value will force the browser (client) to make a new request each time.

## Rules for Caching

- If content is mutable (as most business entities), use [Control-Cache: no-cache](#)
- If content is deemed immutable set a reasonable max-age with [Control-Cache: max-age= 31536000](#)

## Your Turn! - Task 3-1

- Currently, the invoice API returns too many records when retrieving all records (within Task 2-6)
  - Implement **pagination** on the Invoice application
  - Return 50 records at a time for the GET all method
  - Retrieve a **page** argument from the client
  - Use Flask's SQLAlchemy Pagination object
- Work from the provided **task3\_1\_starter.py**

# Rate Limits



Homepage: <https://flask-limiter.readthedocs.io/en/stable/>

- Clients can make too many API requests overwhelming the server

- Applying rate limits can reduce CPU workload and network bandwidth
- Flask provides a 3<sup>rd</sup> party tool (that must be installed), called **Flask-Limiter**

```
pip install flask-limiter  
pip3 install flask-limiter  
pip3.10 install flask-limiter  
python -m pip install flask-limiter
```

```
from flask_limiter import Limiter  
from flask_limiter.util import get_remote_address  
from pathlib import Path  
  
app = Flask(__name__)  
limiter = Limiter(app, key_func=get_remote_address)  
  
@app.route('/api/celebrities/<name>', methods=['GET'])  
@limiter.limit('3/minute')  
def get_one_celebrity(name):  
  
...
```

*student\_files/ch03\_adv\_concepts/05\_rate\_limiting.py*

For more on the Flask rate limiter, visit: <https://flask-limiter.readthedocs.io/en/stable/>.

Different rate limit values can be specified (they are parsed by the framework). Here are a few examples:

- 100 per hour
- 75/hour
- 50 per 2 hours
- 500/day;1000 per 7 days

# API Versioning

- **Versioning** is needed when an API interface changes
  - Versioning refers to the client's perspective of the API
- Versioning *is required* when
  - The API sends back a new data format or values
  - APIs are removed
- Versioning *is not required* when a server *implementation* changes
  - For example, if you change your database implementation from MySQL to MS SQL Server
- Versioning *is also not required* when
  - New endpoints are added
  - New parameters are added to the response (generally)

Clients may spend considerable effort developing their apps to work with your API.

If you change your API, the client will need to update their apps as well. This can take time. To ensure the smoothest transition from the old API version to the new one, for a time, you will need to have both versions available simultaneously.

But how to do this? The answer is versioning. How do we version our API? Several strategies exist and are easy for the client to implement (mentioned momentarily).

# Versioning Approaches

- Versioning can be implemented several ways

- Via the domain name: `http://apiv1.sample.com/api/celebrities`

- Via the path: `http://localhost:8051/api/v1/celebrities`

- Via query string: `http://localhost:8051/api/celebrities?version=1`

- Via custom  
HTTP headers:

`POST /celebrities HTTP/1.1`

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)

Host: www.sample.com

Content-Type: application/x-www-form-urlencoded

...

`Accept-version: v1`

`name=Fred%20Savage&pay=3.0&category=Actors`

The path approach is one of the most common techniques used, particularly for some of the large domain public APIs.

# Defining the Contract and OAS

- RESTful services can be written as **contract-first** or **contract-last**
  - **Contract-first** - define and design the service, then code it
  - **Contract-last** - code up the API, then document it
- One way to create a contract-first solution is by using an Open-API Specification (OAS)
  - OAS uses either YAML or JSON for its format
    - Classic Swagger solutions favor YAML, while OAS proposed a JSON-based format

For more on OAS version 3.x, visit <https://spec.openapis.org/oas/v3.1.0> and <https://swagger.io/specification/>.

# Defining the API with OAS

```
{  
    "openapi": "3.1.0",  
    "info": {  
        "title": "Celebrity App",  
        "summary": "A celebrity info retrieval system.",  
        "description": "This is a partial listing for our celebrity api.",  
        "version": "1.0.0"  
    },  
    "paths": {  
        "/api/celebrities": {  
            "get": {  
                "description": "Returns all celebrities.",  
                "responses": {  
                    "200": {  
                        "description": "A list of celebrities.",  
                        "content": {  
                            "application/json": {  
                                "schema": {  
                                    "type": "array"  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        },  
        "/api/v2/celebrities": { ...  
    }  
}
```

*student\_files/ch03\_adv\_concepts/openapi.json*

This shows only a partial listing of what an OAS document structure looks like. It can be somewhat lengthy identifying numerous other items, such as servers involved, all of the operations and various paths, security information, and much more. It can also maintain multiple API versions (partially shown here).

## Auto Generating an API

- Tools (for many languages) exist to generate code from an OAS document for both the API server and client

<https://openapi-generator.tech/>

Select "Generators"

- **Swagger Codegen** provides the ability to create an API based on an OpenAPI document
- **Connexion**, is a framework that generates an API server that maps your functions to your endpoints from a JSON or YAML-defined document (see next example)

For more on OAS version 3.x, visit <https://spec.openapis.org/oas/v3.1.0> and <https://swagger.io/specification/>.

# Generating a Solution with Connexion (1 of 3)

- The following runs Connexion to create our auto-generated API interface

**api.py**

```
def get_celebrity(name):
    return f'You selected: {name}', 200
```

*To run this:*

- 1) Go to ch03\_adv\_concepts/connexion\_example
- 2) Run connexion\_sample.py
- 3) Navigate to: <http://localhost:8051/ui/>

**connexion\_sample.py**

```
import connexion

from connexion.resolver import RestyResolver

app = connexion.App(__name__)
app.add_api('swagger.yaml', resolver=RestyResolver('api'))

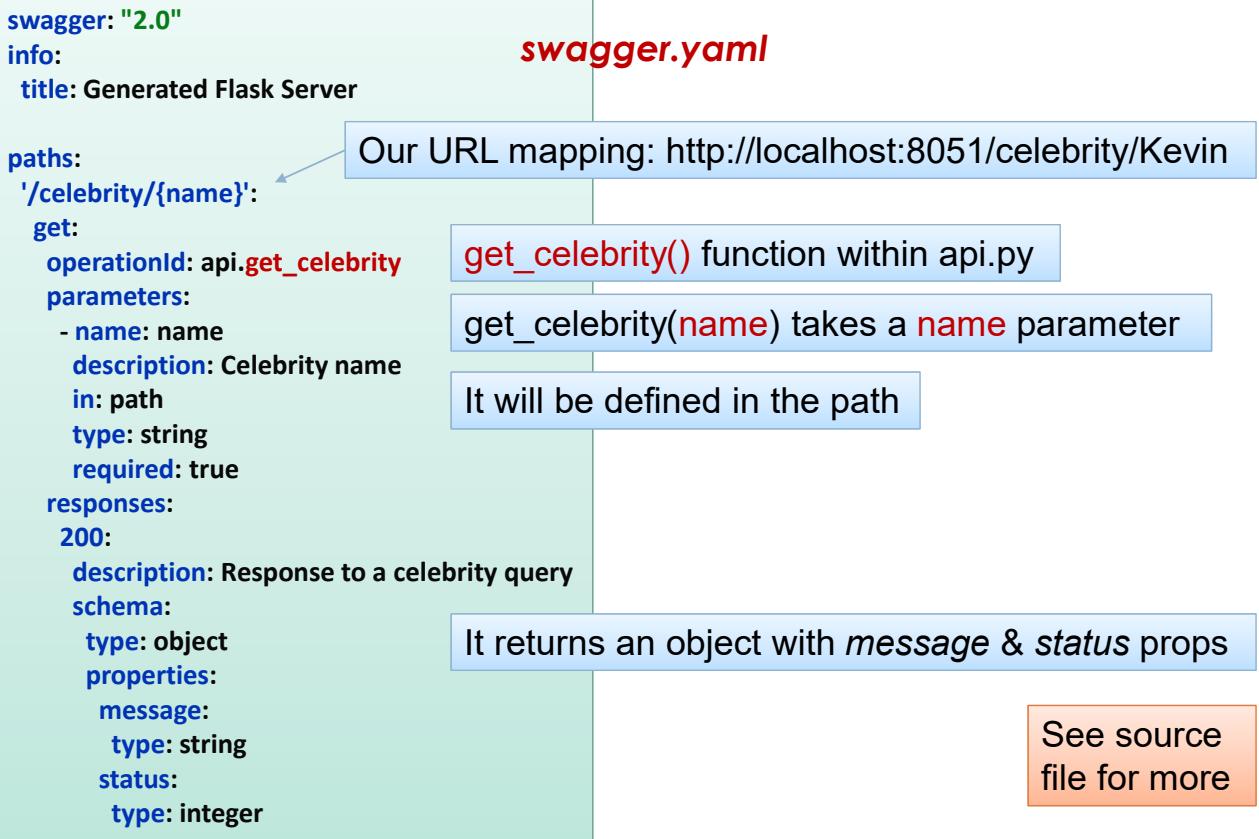
app.run(host='localhost', port=8051)
```

api.py

*ch03\_adv\_concepts/connexion\_example*

In order to successfully run this, you must have run the Task 1-1 setup at the start of our course. This will install necessary tools such as Flask and Connexion[swagger-ui].

# Generating a Solution with Connexion (2 of 3)



`ch03_adv_concepts/connexion_example`

The first line of the document defines the format of the content. If it is `swagger: "2.0"` it will have one structure other wise it could be `openapi: "3.0.0"` which will define a slightly different schema.

# Generating a Solution with Connexion (3 of 3)

The screenshot shows the Swagger UI interface for a Connexion-generated Flask server. At the top, there's a green header bar with the 'swagger' logo, the URL 'http://localhost:8051/swagger.json', and a 'Explore' button. Below the header, the title 'Generated Flask Server' is displayed, along with the subtitle 'Example Generating Flask Server From Swagger YAML'. To the right, a callout box contains the text 'With the server running, navigate to: http://localhost:8051/ui'. The main content area is titled 'default' and lists various API operations:

| Method | Path              |
|--------|-------------------|
| GET    | /celebrity        |
| POST   | /celebrity        |
| DELETE | /celebrity/{name} |
| GET    | /celebrity/{name} |
| PUT    | /celebrity/{name} |

Below the table, a note states '[ BASE URL: , API VERSION: 1.0.0 ]'. A callout box highlights the five rows of the table with the text 'Each of these provides a "Postman-style" interface to test with'.

*ch03\_adv\_concepts/connexion\_example*

Shown here is the autogenerated client for the Connexion example.

## HATEOAS

- One problem with REST (or any service architecture) is that documentation is critical
- Every client must ask, "What are the valid URLs for this RESTful service?"
- To *promote looser coupling* between client and server and to provide a means for clients to *automatically discover* these services (resources), HATEOAS comes in
  - Stands for: Hypermedia As The Engine of Application State

To help promote loose coupling between the client and server (i.e., so that the client doesn't have to hard-code or have knowledge of various resource links), related resources can be embedded into GET requests. This can be done in a "links" property of the GET JSON response. The HATEOAS format is up to you; it doesn't have to be structured exactly like the example.

For more on this topic, check out: <https://restcookbook.com/Basics/hateoas/>

# Sample HATEOAS Response

```
{  
    "InvoiceNo": "536365",  
    "StockCode": "85123A",  
    "Quantity": 6,  
    "Description": "LIGHT-HOLDER",  
    "UnitPrice": 2.55,  
    "CustomerID": "17850",  
    "Country": "United Kingdom",  
  
    "links": [  
        {  
            "rel": "customer",  
            "href": "https://server.com/customers/17850",  
            "action": "GET",  
            "types": ["text/xml", "application/json"]  
        },  
        {  
            "rel": "customer",  
            "href": "https://server.com/customers/17850",  
            "action": "PUT",  
            "types": ["application/x-www-form-urlencoded"]  
        },  
        ... (see example in student files for more)...  
    ]  
}
```

Server's GET Response

A client must know the first request to make, but subsequent requests can be "discovered" in the "links"(or equivalent) section of the server's response

*student\_files/ch03\_adv\_concepts/hateoas.json*

There are different formats for performing discovery. HATEOAS is only one.

Also, in the works, is a way to define better support with JSON for linked actions. HAL and JSON-LD are two specifications that are evolving for this purpose.

## Chapter 3 Summary

- Flexible APIs can provide filtering, sorting, and pagination features
  - These are commonly implemented using query string parameters
  - While a database might prefer limit and offset, the client doesn't need these
    - Simply provide a page attribute to allow pagination
- Caching can reduce the workload for both the API server and client
  - Client caching is easily controlled through Cache-Control
  - Other headers exist but often aren't necessary for APIs

# Chapter 4

## Authorization and Authentication



# Chapter 4 Overview

## **Security & RESTful APIs**

Security Options

JWT

OAuth 2

# Securing RESTful APIs

- Most APIs require a level of security before resources can be accessed
- Several actively used authentication techniques exist
  - Basic Authentication
  - OAuth 1.0 (Digest Authentication)
  - Token (Bearer) Authentication (e.g., JSON Web Tokens, JWT)
  - API Keys
  - OAuth 2.0

There are several approaches that can be used when securing APIs. We'll discuss these in this chapter.

## Basic Auth

- Basic authentication can be used as a challenge against unwanted visitors
  - It requires the use of a secure channel as credentials must be submitted in the HTTP request
  - Credentials are only encoded not encrypted so it should only be used with HTTPS

```
import requests
from requests.auth import HTTPBasicAuth
from bs4 import BeautifulSoup

page = 'https://jigsaw.w3.org/HTTP/Basic/'
auth = HTTPBasicAuth('guest', 'guest')
page_text = requests.get(page, auth=auth).text
soup = BeautifulSoup(page_text, 'html.parser')
print(soup.select('p')[2].text)
```

Python requests can handle basic auth queries

student\_files/ch04\_auth/01\_basic\_auth.py

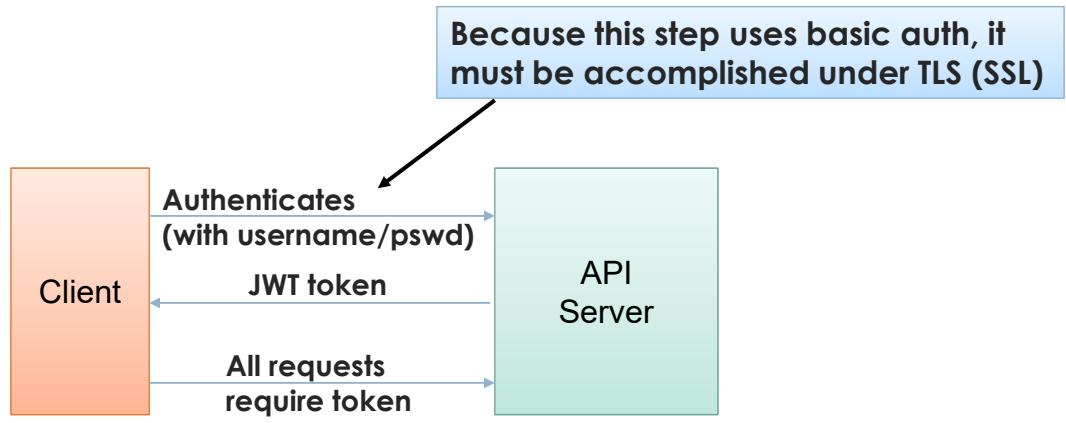
Basic authentication is secure only if you always make sure to use TLS (SSL).

In addition, basic auth always requires username and password to be sent and therefore raises the value of that request. In other words, if we are using OAuth 2, we are only sending a token each time. Compromise of that token while problematic, only gives the attacker access to that session, not the user's credentials or account info.

Compromise of a request that uses basic authentication exposes the user's username and password and potentially invalidates that user's account, possibly more. So careful consideration should be given to using basic auth in applications. Consideration as to how credentials are initially provided (like a web-based form, for example) or the cost to the user if the requests are compromised will need to be examined.

# JSON Web Token-Based Authentication

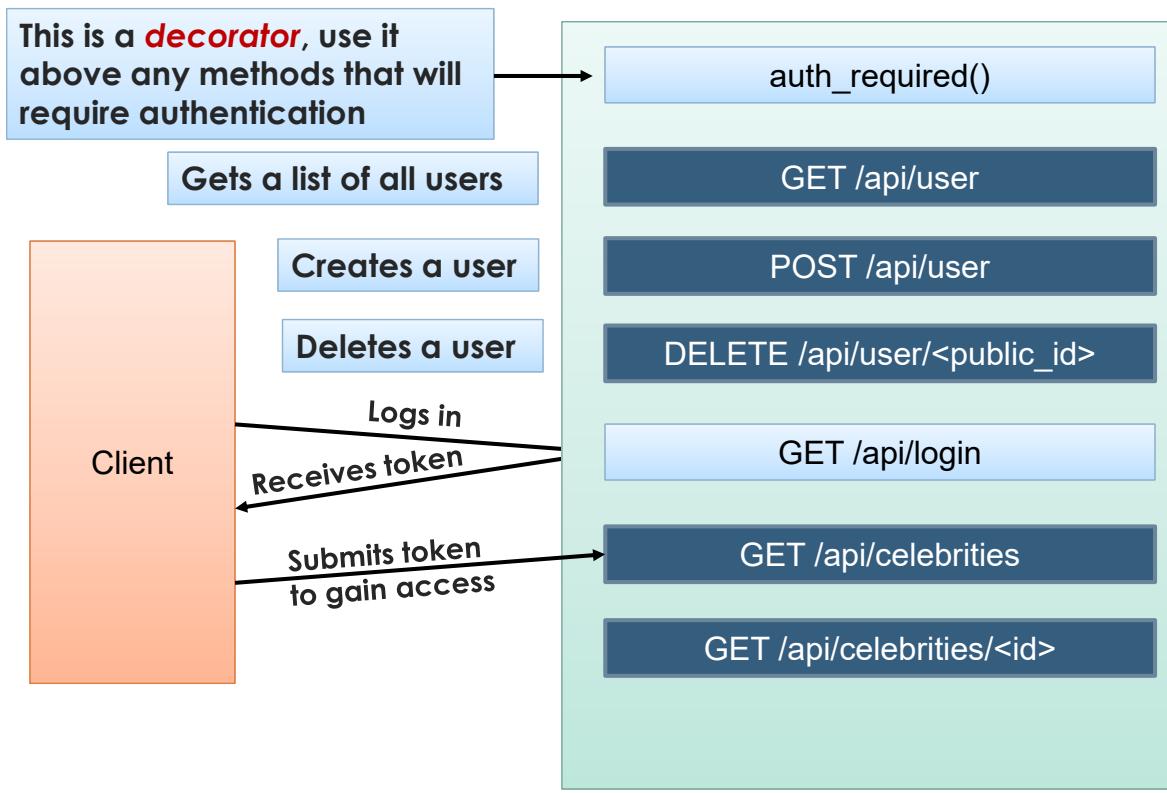
- JSON Web Tokens (JWTs) can be a useful means for securing APIs



- JWTs only require authentication once, after that a token is used for authentication and authorization

JSON web tokens provide a means for identifying a user by examining a provided token. Because tokens are signed, they can't be tampered with by intercepting them. Because of the way the tokens are created, they can't be invalidated. They can only be allowed to expire.

# JWT Token-Based Authentication



`student_files/ch04_auth/02_celeb_auth.py`

Our example exists in two files. The server is `02_celeb_auth.py` while the client is `03_test_celeb_auth.py`.

Our API has two sets of interfaces. One interface manages users while the other allows access to celebrity info.

# The *auth\_required* Decorator

```
def auth_required(orig_func):
    def wrapper(*args, **kwargs):
        token = None
        if 'x-access-token' in request.headers:
            token = request.headers['x-access-token']

        if not token:
            return jsonify({'message': 'Token is missing!'}), 401

        try:
            data = jwt.decode(jwt=token,
                              key=app.config['SECRET_KEY'],
                              algorithms=['HS256'])
            current_user = User.query.filter_by(public_id=data.get('public_id')).first()
        except Exception as err:
            print('Token is invalid-->', err)
            return jsonify({'message': 'Token is invalid!'}), 401

        return orig_func(current_user, *args, **kwargs)
    return wrapper
```

The decorator is placed above any functions requiring authentication

Failure to authenticate will result in messages regarding the token

If the token is present, it is decoded using the jwt.decode() method

The decoded token should contain a public id of the user, which is used to query the database to obtain the user (if they exist)

*student\_files/ch04\_auth/02\_celeb\_auth.py (server)*

A little background on decorators can be found in the appendix.

This decorator should be placed above any function that will require authentication. It will look for the presence of the token under the key of "x-access-token" header. If it is not found or incorrect, a message will be returned. Otherwise, if it is present, it will be decoded using the jwt module's decode() method. The decoded token will result in a dictionary containing the user's public\_id. This can be queried in the database to obtain the user object. With the user object, the originally desired function will be called and the current\_user object will be injected into it. So, all functions that are decorated by this decorator must provide *current\_user* as the first argument.

# Testing JWT API Authentication

```
login_response = requests.get(f'{base_url}{path}/login', auth=credentials).json()  
token = login_response.get('token')
```

The client logs in with credentials  
and gets a token in return

The token is added into the headers  
on each subsequent request

```
r = requests.get(f'{base_url}{path}/celebrities', headers={'x-access-token': token}).json()
```

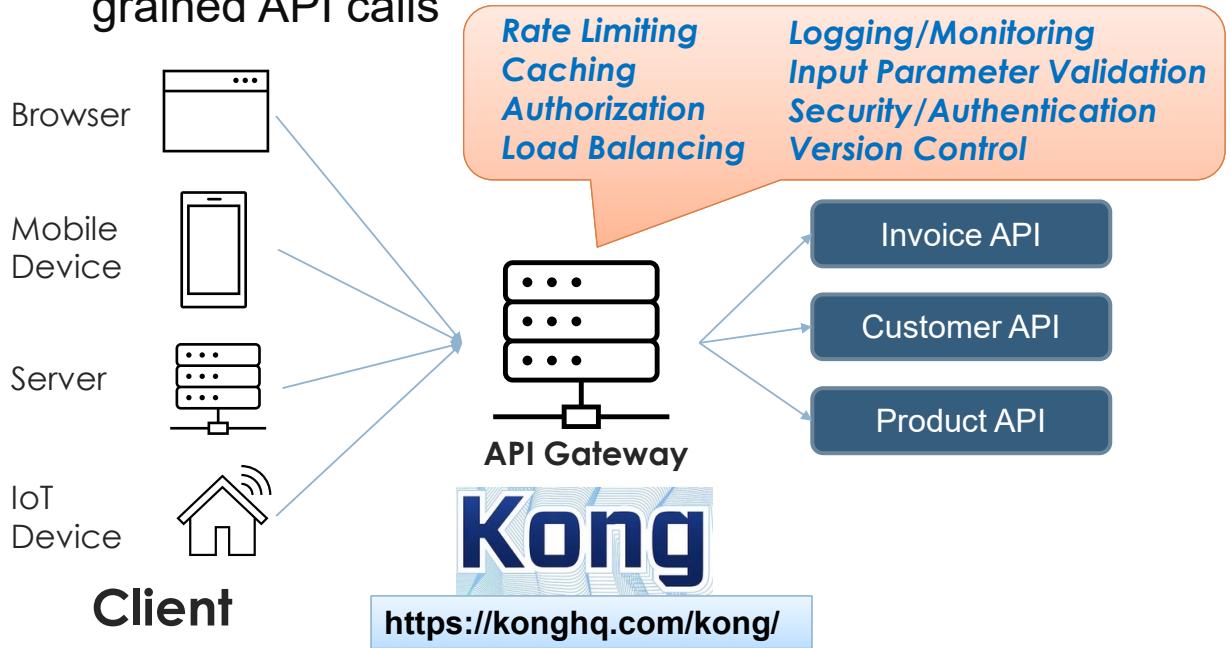
The test client (not shown) performs the following steps:

- Access celebrities without a token
- Access celebrities with an invalid token
- Authenticate (/api/login) as an admin
- Receives a token
- Access celebrities (with token) successfully
- Creates a new user (Sally) (not an admin)
- Logs in with the new user
- Retrieves a single celebrity successfully
- Attempts to delete a user (fails)
- Log in as admin user
- Deletes Sally

*student\_files/ch04\_auth/03\_test\_celeb\_auth.py (client)*

# API Gateway

- An API Gateway serves as a single point of entry when a client may need to interact with numerous fine-grained API calls

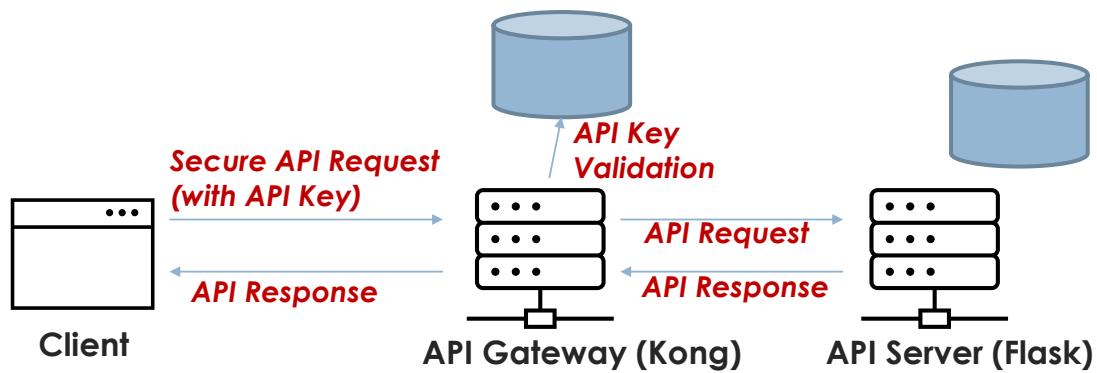


API gateways assist with providing a single interface in front of a microservices architecture thus simplifying client knowledge of the API network.

There are numerous options now for open-source API gateways. While KONG is the most popular and well-known, others such as Apache APISIX, Tyk (AWS), and Ocelot (for .NET) also exist.

## API Keys

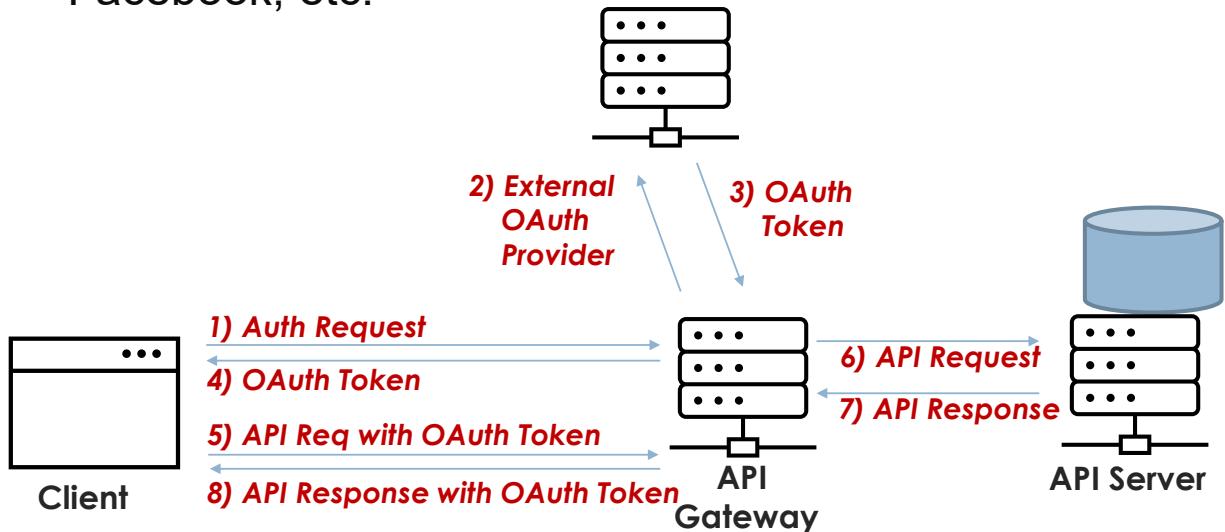
- API keys provide a means to link API access to a specific client
  - Usually, a client registers for an API key
  - The key is submitted for every GET request which identifies the client
  - Resource access and rate limits can then be applied
  - Common within public (non-private) APIs
  - API keys are not suitable for POST, PUT, PATCH, or DELETE



The use of an API key has become common particularly with publicly visible API services. Keys alone are typically not used to secure services. Instead, they are used as a tracking mechanism to control access and limit usage of the API.

# OAuth 2

- OAuth provides token-based identity verification services from an external provider such as Google, Facebook, etc.



OAuth 2 is an authentication protocol that allows a client to communicate with a server once the client has been verified by a designated third-party provider. Once the provider verifies the client, it sends a token back to the client (via the API gateway or server) to be used for all subsequent requests made to the API host. OAuth 1 and OAuth 2 are completely different from each other.

Using an OAuth verifier provides easy access to services without the need to pass credentials. OAuth tokens must be provided in order to gain access to the API server. Tokens have an expiration (usually an hour).

## Security Best Practices

- APIs should always send and receive over SSL (TLS)
- APIs that allow POST, PUT, PATCH and DELETE should use stronger forms of authentication, such as OAuth 2 or OpenID Connect
- Place API keys in the **Authorization:** header instead of the query string (as is too commonly done)

Authorization: Bearer zY%^33GhibW2%11Qy#1a8Z

## Chapter 4 Summary

- As a minimum, all APIs should implement transport security using TLS (SSL)
- Use API keys for public APIs that primarily support GET operations to monitor client usage metrics
- Apply stronger forms of authentication for full-service APIs
  - Use OAuth 1 or 2 for most needs

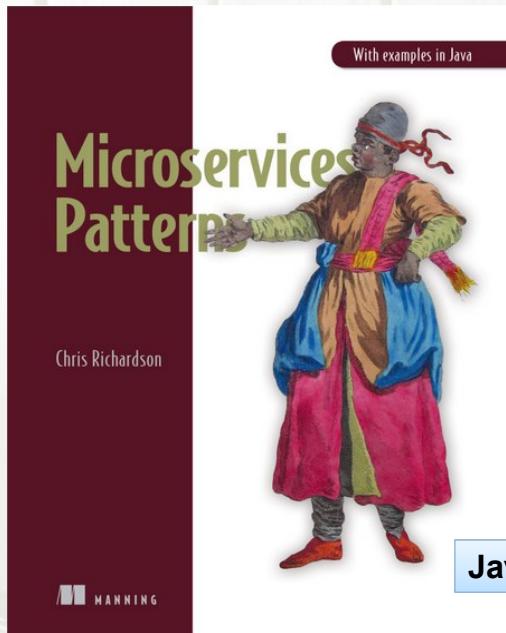
# Course Summary

## **What Did We Learn?**

- |                                |                                      |
|--------------------------------|--------------------------------------|
| <b>REST vs Web Services</b>    | <b>Flask-Marshmallow</b>             |
| <b>JSON Syntax</b>             | <b>PUT vs PATCH</b>                  |
| <b>Key REST Components</b>     | <b>Error Handling</b>                |
| <b>Design Principles</b>       | <b>Pagination</b>                    |
| <b>RESTful Methods</b>         | <b>Caching</b>                       |
| <b>Endpoints and Resources</b> | <b>Rate-Limits</b>                   |
| <b>Python Requests</b>         | <b>Versioning</b>                    |
| <b>Flask / Flask-RESTX</b>     | <b>Data Compression</b>              |
| <b>Routes</b>                  | <b>HATEOAS</b>                       |
| <b>Postman</b>                 | <b>Options for Securing Services</b> |
| <b>Swagger</b>                 | <b>OAuth</b>                         |
| <b>Database Integration</b>    | <b>API Keys</b>                      |
| <b>Flask-SQLAlchemy</b>        | <b>Considerations for Security</b>   |
| <b>Object Serialization</b>    |                                      |

## Recommended Sources

Covers Flask, Django, SQLAlchemy, Other Python Tools, slightly dated (2018)



Java-based, but patterns are patterns (2018)

## *Other Resources*

### **Well-known Flask Tutorial (2017)**

<https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

### **REST API with Flask & SQLAlchemy**

<https://www.youtube.com/watch?v=PTZiDnuC86g>

### **Awesome API - Useful collection of resources**

<https://github.com/Kikobeats/awesome-api>

udemy

Development > Web Development > Flask

### **REST APIs with Flask and Python**

Build professional REST APIs with Python, Flask, Flask-RESTful, and Flask-SQLAlchemy

<https://www.udemy.com/course/rest-api-flask-and-python/>

## Questions

# Questions?

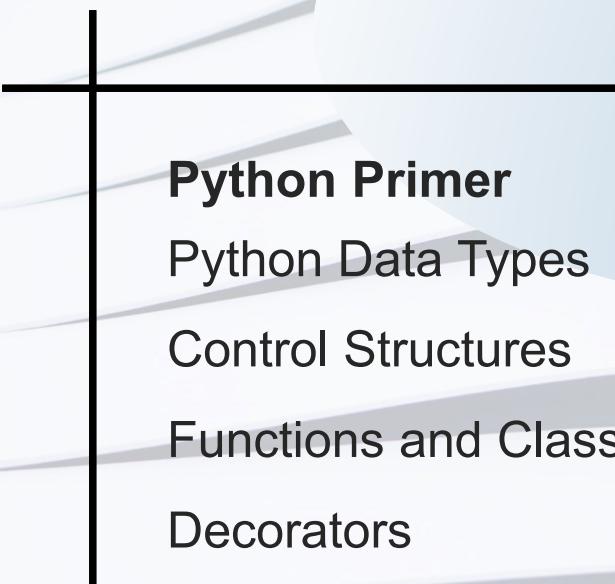


# Appendix

# Python Primer

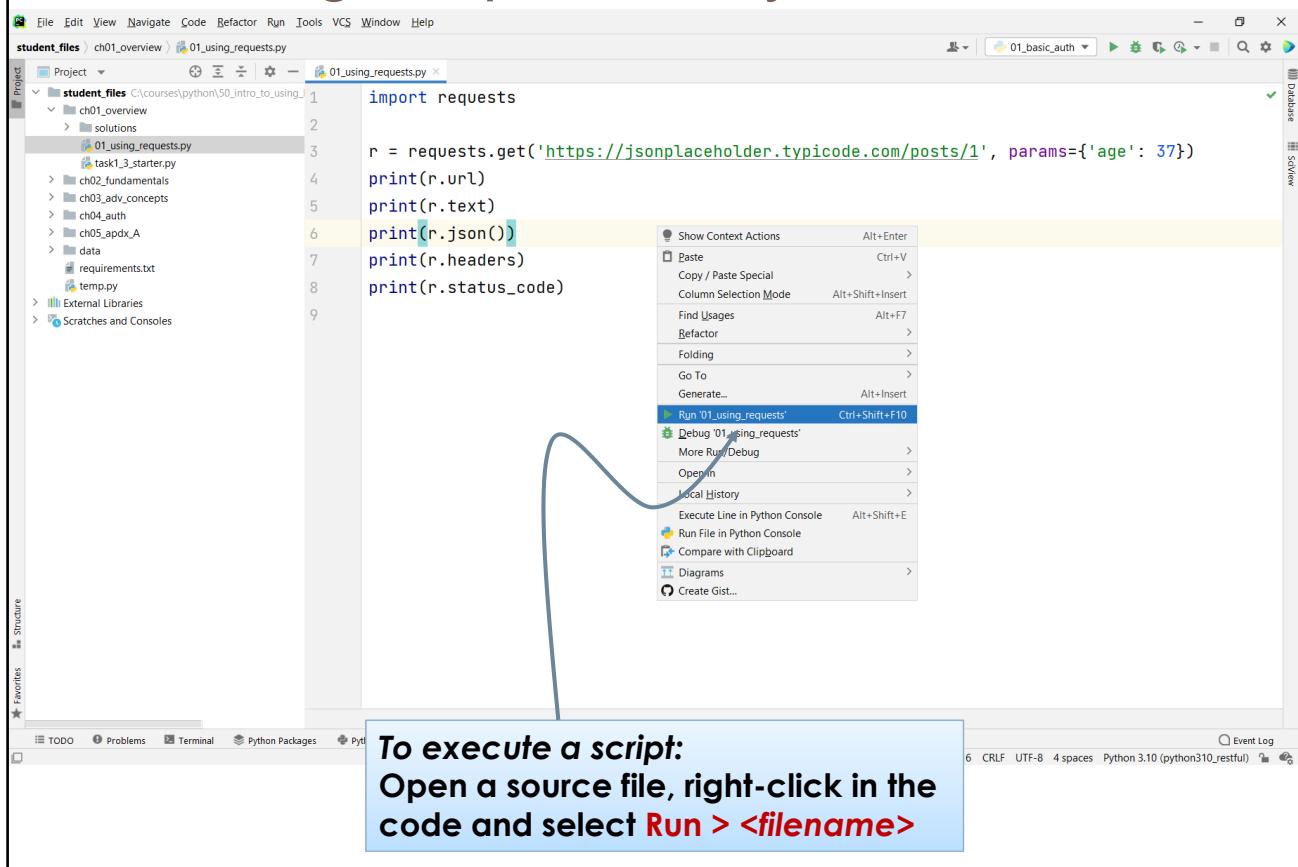


# Appendix Overview



**Python Primer**  
Python Data Types  
Control Structures  
Functions and Classes  
Decorators

# Running Scripts with PyCharm



To work with PyCharm, launch the program and select Open... from the options on the right and then locate the student\_files directory.

Usually, a good first step is to verify the correct interpreter is selected for use. Do this by visiting File > Settings > Project: student\_files > Project Interpreter and examining the interpreter option.

To execute a script, you will first need to wait until the IDE has "scanned" the environment. You may see a message "3 processes running" in the footer. Wait until this completes, then attempt to run the script.

# Providing Command-Line Arguments

- When a script is run from a command-line, additional arguments can be retrieved
  - Import the `sys` module, use the `argv` list (array)

```
import sys  
  
pet1, pet2 = 'dog', 'cat'  
  
if len(sys.argv) >= 3:  
    pet1 = sys.argv[1]  
    pet2 = sys.argv[2]  
  
print('My', pet1, 'chases my', pet2)
```

- When running this script:

`python example01.py`

`python example01.py cat bird`

argv[0] argv[1] argv[2]

My dog chases my cat

My cat chases my bird

*student\_files/ch05\_apdx\_A/example01.py*

# Python Data Types

- Python has many built-in data types

```
x = 10
print(x)          # 10
print(type(x))   # <class 'int'>

x = 'hello'
print(x)          # hello
print(type(x))   # <class 'str'>

type(x) is str    # True
isinstance(x, str) # True
```

A variable is created when you first assign it a value

Variable type values can change (types are checked only at runtime)

- `type()` provides insight about what *kind* a variable is at runtime
  - Use `isinstance()` in practice to verify a variable's type

# Converting Values

- Sometimes values need to be converted from one format to another
  - int or float *to String (str)*

```
s1 = str(55)          # '55'  
s2 = str(3.14)        # '3.14'
```

- String or float *to int*

```
i1 = int('37')        # 37  
i2 = int(3.14)         # 3
```

- int or String *to float*

```
f1 = float(55)        # 55.0  
f2 = float('3.14')     # 3.14
```

Values that can't be converted will generate an exception called a *ValueError*

`int('hello')` yields...

```
ValueError: invalid literal for int() with base 10: 'hello'
```

Exceptions such as ValueErrors can be properly dealt with using exception handling (not discussed in this primer).

# Strings

- Strings are immutable **sequences** of Unicode characters

- Type: **str**

- Formal notation:

```
my_str = str('Python is great!')
```

```
my_str = 'Python is great!'
```

```
my_str = 'Python is fun'
```

```
my_str = 'Python is very fun'
```

Creates a new string object

- Strings support both single and double quotes

PEP 8 does not have a preference

Triple quoted strings can span multiple lines. A common use for triple quoted strings is for documentation (called a docstring).

```
my_str = 'Python\'s great fun'
```

```
my_str = "Python is great fun"
```

```
my_str = """Python is so much fun"""
```

In Python 3, strings are Unicode characters. Some characters must be escaped to use them. Escape characters include:

|     |                                      |
|-----|--------------------------------------|
| \\\ | backslash                            |
| \'  | single quote (as shown in the slide) |
| \\" | double quote                         |
| \b  | backspace character (ascii)          |
| \n  | linefeed                             |
| \r  | carriage return                      |
| \t  | tab                                  |

# Conditional Control Structures

- Python has only one conditional control

There may be zero or more `elif` branches

The `else` is optional

Beginning and ending of blocks are determined by indentations (use 4 spaces)

```
if test:  
    <one or more statements>  
elif test2:  
    <one or more statements>  
else:  
    <one or more statements>
```

```
if test:  
    print('If test is true, this will execute')  
    print('So will this!')  
print('This will always execute!')
```

# Sequences

- In Python, sequences (**str**, **list**, and **tuple** types) are ordered collections of objects

```
dirs = ['North', 'South', 'East', 'West']
```

- All sequences have some common features:

- Random access and slicing

```
dirs[2] # 'East'
```

```
dirs[-2:] # ['East', 'West']
```

- Concatenation  
(of the same type)

```
dirs + ['NW', 'NE', 'SW', 'SE']
```

- Length (size)

```
len(dirs), len(dirs[0])
```

- Membership

```
if 'East' in dirs:  
    print(dirs.index('East'))
```

Sequences exhibit similar behaviors such as the ability to be randomly accessed, perform slicing, support membership checks, and have their size (length) returned through the `len()` function.

In addition, sequences also support a `min()` and `max()` function.

Examples of `min()` and `max()`:

```
max([1973, 2001, 2015, 2013, 1994])
```

```
min([1973, 2001, 2015, 2013, 1994])
```

# Lists Are Python's Arrays

- Lists are mutable, ordered collections of objects

```
my_list = []
```

```
my_list = list()
```

```
my_list = [1, 3, 5]
```

```
my_list = [3.3, 'hello', Person()]
```

```
my_list = list('hello')
```

Empty lists

List created using the type name

some\_list = list(other\_sequence)

- Lists may contain duplicates

```
my_list = [3.3, 'hello', Person(), 3.3, Person()]
```

- Adding items to a list

```
my_list = [1, 2, 3]
```

```
my_list.append(10)
```

```
my_list.insert(1, 'hello')
```

1, 'hello', 2, 3, 10

*student\_files/ch05\_apdx\_A/example02.py*

Lists may be created using literal notation with square brackets [ ]. While literal notation is common, occasionally lists are created using the list() type name.

Use append() to add items onto the end of the list.

To add a value into the middle of a list, use insert(position, value), keeping in mind that sequences are zero-based in Python.

# Tuples Are Python's Fixed Arrays

- Tuples are **immutable**, ordered collections of objects

```
my_tuple = ()
```

Empty tuple

```
my_tuple = tuple()
```

Tuple created using the type name

```
my_tuple = (1, 3, 5)
```

```
my_tuple = (3.3, 'hello', Person())
```

```
some_tuple = tuple(sequence)
```

```
my_tuple = tuple('hello')
```

Tuple with one element

```
my_tuple = (1,)
```

- Modifying a tuple causes an error, like this:

```
contact = ('John', 'Smith', ['123 Main St', 'Los Angeles', 'CA'])
```

```
contact[0] = 'Jonathan'
```

TypeError: 'tuple' object does not support item assignment

Tuples do not have append(), insert(), or other methods that attempt to modify the data as this would violate the concept of a tuple.

# Iterative Control Structures

```
while test:  
    <one or more statements>
```

This will execute the contents of the indented block as long as test is True

```
for one_item in iterable:  
    <one or more statements>
```

This will take one item from the iterable at a time, process it within the indented block, and then get the next item.

- Performance note:
  - For larger iterations, the for-loop performs better than the while-loop and should usually be preferred
  - *for* executes at the C-level, *while* executes in the PVM

Note: Both the while and for loops have a second part, an else block. However, because this block is extremely rare to use, it has been left out of our discussion here.

## Examples Using the *for* Loop

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']

for season in seasons:
    if season.lower().startswith('s'):
        print(f'{season} has {len(season)} characters.')
```

Spring has 6 characters.  
Summer has 6 characters.

```
records = [
    ('John', 'Smith', 43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James', 32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer', 29, 'kcramer@sintech.com')
]

for record in records:
    print(f'{record[0]} {record[1]}, {record[2]} {record[3]}'")
```

John Smith, 43 jsbrony@yahoo.com  
Ellen James, 32 jamestel@google.com  
Sally Edwards, 36 steclone@yahoo.com  
Keith Cramer, 29 kcramer@sintech.com

*student\_files/ch05\_apdx\_A/example02.py*

Python's for and while loops are the only iterative control structures. There is not a classic 3 part for( ) loop as in C++, Java, and other languages. Both while and for support the break and continue statements.

## Iterating (*continued*)

- All of these can iterate the *records* data structure

*student\_files/ch05\_apdx\_A/example02.py*

Each of these produces the same output as the one on the previous slide. The last, while a bit cumbersome, can be useful when the string is a variable that's already defined elsewhere and can't be changed.

# Primer on Dictionaries

- Dictionaries are collections of name/value pairs
  - They are unordered, mutable, iterable
  - Support `len()` function and `in` operator
  - Keys are hashable (immutable), values can be any type

```
d = { key1: value1, key2: value2, ... }
```

```
my_dict = {}  
my_dict = dict()  
my_dict = { 'pet1': 'dog', 'pet2': 'fish' }  
my_dict = dict(pet1='dog', pet2='fish')  
  
my_dict['pet3'] = 'cat'  
print(my_dict['pet2'])
```

empty dicts

No quotes on keys here

adding / changing values

accessing values

Dictionaries are useful data structures as they provide a means to store any kind of data yet access that data via a key. Because dictionary keys must be unique, attempting to add an entry to a dictionary that has the same key in existence will replace the value with the new value.

Dictionaries became ordered in Python 3.6.

# Accessing Dictionaries

```
d1 = {'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29}
```

- Direct access can generate a **KeyError**

|                          |                               |
|--------------------------|-------------------------------|
| <code>d1['Smith']</code> | <i># returns 43</i>           |
| <code>d1['Green']</code> | <i># generates a KeyError</i> |

- Use exception handling to deal with a **KeyError**

```
try:
    value = d1['Green']    # generates a KeyError
except KeyError:
    value = 0
```

- `dict.get(key)` to retrieve values also:

|                                |                       |
|--------------------------------|-----------------------|
| <code>d1.get('Edwards')</code> | <i># returns 36</i>   |
| <code>d1.get('Green')</code>   | <i># returns None</i> |

- `dict.get(key, default)` is safest

|                                  |                     |
|----------------------------------|---------------------|
| <code>d1.get('Cramer', 0)</code> | <i># returns 29</i> |
| <code>d1.get('Green', 0)</code>  | <i># returns 0</i>  |

*student\_files/ch05\_apdx\_A/example03.py*

Accessing a dictionary in a for loop will return the keys. It is the same effect as `d1.keys()`, which is arguably more understandable but also more verbose.

# Iterating Dictionaries

```
d1 = {'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29}
```

- Iterating a dictionary directly returns keys:

```
for item in d1:  
    print(item)
```

Smith  
James  
Edwards  
Cramer

- Iterating a dictionary's values:

```
for val in d1.values():  
    print(val)
```

43  
32  
36  
29

- Accessing both key and value simultaneously:

```
for key, val in d1.items():  
    print(f'Key: {key}, Value: {val}')
```

returns a (view of) list of tuples

*student\_files/ch05\_apdx\_A/example03.py*

The above example indicates how a dictionary is used within a `for` control. It always returns the keys.

The `items()` method returns a "view" object instead of a copy of a list of tuples.

# Defining and Calling Functions

- Functions must be defined before they can be called

```
def summary(customer: dict, amount: float = 0.0):  
    return f'Customer: {customer.get("first")}\n{customer.get("last")}, amount: ${amount:.2f}'
```

Function statements  
must be indented

Type hints

Default argument

Return values are optional. A  
value of 'None' is returned when  
a return statement is omitted

```
cust = {  
    'first': 'James',  
    'last': 'Smith'  
}
```

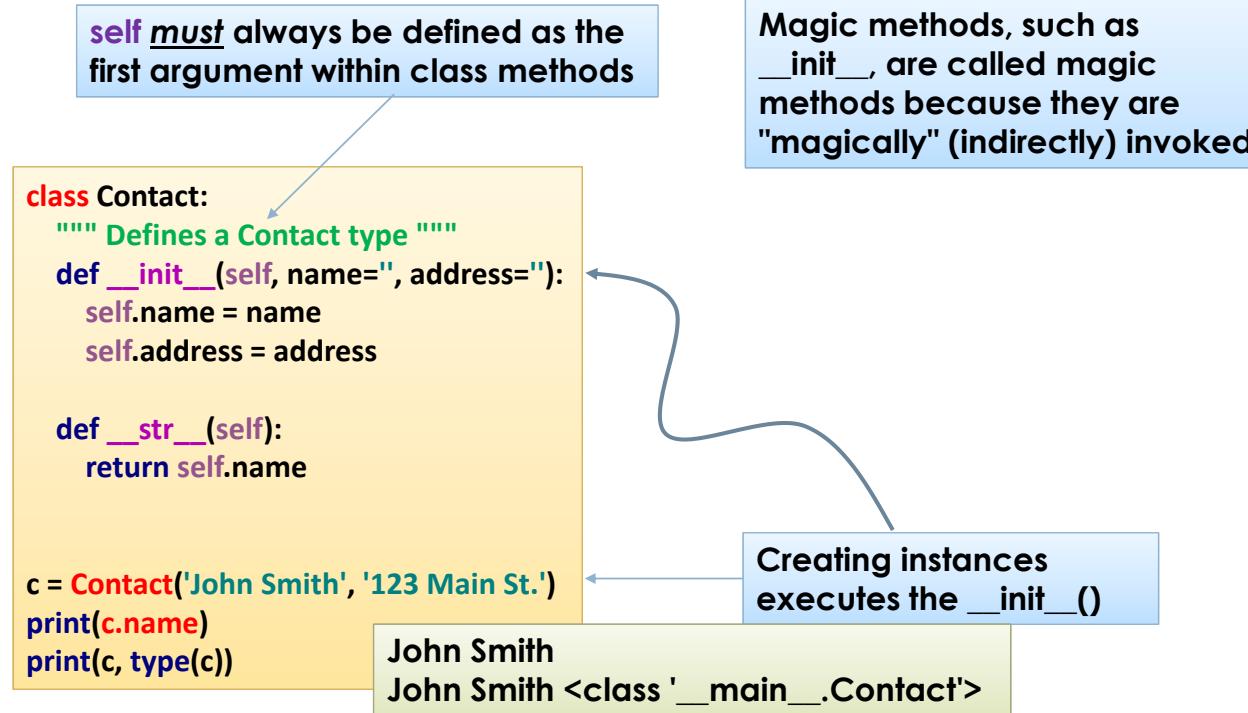
```
results = summary(cust, 1108.23)  
print(results)
```

Customer: James Smith, amount: \$1,108.23

*student\_files/ch05\_apdx\_A/example04.py*

Functions must be declared (or imported) before they can be called. Parameters passed into the function must also match what is declared in the function.

# Classes



`student_files/ch05_apdx_A/example05.py`

When an instance (object) is created the constructor is called. In Python, the constructor is always called. Its purpose is to initialize variables and/or code.

The `__init__()` (constructor essentially) must always define a variable called `self`. `Self` represents the current object being constructed. In this example, the object called 'c' is being created. In the constructor, the 'c' object is what `self` is referring to. While theoretically `self` can be renamed to something else, in Python we **never** do this.

## Decorators (*in brief*)

```
def short_formatter(func):
    width = 15
    def wrapper(val):
        val = val[:width] + '...'
        func(val)
    return wrapper
```

```
@short_formatter
def display(val):
    print(val)
```

```
data = 'This is a long string that will be truncated.'
display(data)
```

This function returns another (nested) function

The returned function replaces the decorated function below (display)

This notation is what causes the display function to be replaced by the one returned from the decorator

*student\_files/ch05\_apdx\_A/example06.py*

In the example above, the original function (display) is passed into another function called short\_formatter(). Short\_formatter() returns a nested function called wrapper(). The returned wrapper function replaces the originally passed function (display). Now, any calls to display() are actually calls to wrapper(). Decorators give the library/framework builder control over your code.

# Appendix B

## Pydantic, FastAPI, and Deployment



# Appendix Overview

Pydantic

FastAPI

Deploying

# Pydantic

- Pydantic is a popular third-party **data validation framework** that checks data values against annotations at runtime

```
from pydantic import BaseModel

class Celebrity(BaseModel):
    name: str = None
    salary: float = 0.0
    year: int = None
    type: str = None

data = ['Oprah Winfrey', 225.0, 2005, 'Personalities']
c = Celebrity(name=data[0], salary=data[1], year=data[2], type=data[3])
_____
headers = Celebrity.model_fields
c = Celebrity(**dict(zip(headers, data)))
_____
data = ['Oprah Winfrey', '225.0', '2005', 'Personalities']
c = Celebrity(name=data[0], salary=data[1], year=data[2], type=data[3])
_____
data = ['Oprah Winfrey', '225.0', 'some year', 'Personalities']
c = Celebrity(name=data[0], salary=data[1], year=data[2], type=data[3])
```

**Pydantic models *must* use keywords to instantiate**

**Data values are coerced in Pydantic**

If a value is not valid, it will generate a `pydantic.ValidationError`

ch06\_apdx\_b/01\_pydantic.py

## FastAPI

- **FastAPI** is a Python-based application server designed specifically to support, high-performance APIs
  - It's Open API compliant
  - Capable of generating schema automatically
  - Compatible with Pydantic Models
  - Conforms to the ASGI specification

# Celebrity with FastAPI

```
import uvicorn
from fastapi import FastAPI
from pydantic import BaseModel

class Celebrity(BaseModel):
    name: str = None
    salary: float = 0.0
    year: int = None
    type: str = None

datafile = Path(__file__).parents[1] / 'data/celebrity_100.csv'
with datafile.open(encoding='utf-8') as f:
    #create list of Celebrity objects

app = FastAPI()

@app.get('/api/celebrities/{name}', response_model=list[Celebrity])
def get_celebs(name: str):
    matches = [celeb for celeb in celebs
               if name.lower() in celeb.name.lower()]
    return matches

uvicorn.run(app, host='127.0.0.1', port=8000)
```

ch06\_apdx\_b/02\_fastapi\_celeb.py

A simple FastAPI example with our celebrity dataset and implementing just the GET method (without any IDs for the celebrity records).

# All API Methods with FastAPI (1 of 2)

```
from fastapi import FastAPI, HTTPException, Request, status
from pydantic import BaseModel

class Celebrity(BaseModel):
    celeb_id: int = None
    name: str = None
    salary: float = 0.0
    year: int = None
    type: str = None

app = FastAPI()

@app.get('/api/celebrities/{celeb_id}')
def get_celebs(celeb_id: int):
    matches = [celeb for celeb in celebs if celeb_id == celeb.celeb_id]
    if not matches:
        raise HTTPException(404, detail='No matching items.')
    return matches[0]

@app.post('/api/celebrities', status_code=status.HTTP_201_CREATED)
def create_celeb(celeb: Celebrity):
    global id_count
    id_count += 1
    celeb.celeb_id = id_count
    celebs.append(celeb)
    return celeb
```

ch06\_apdx\_b/03\_fastapi\_celeb\_full.py

Reads the data into a list and doesn't use a database. But this example illustrates implementing the GET, POST, PUT, and DELETE methods within FastAPI.

In the POST method, since we aren't using an actual database, unique IDs are generated for new records by just using a global `id_count` variable. In the GET method, we can check for conditions and raise a `fastapi.HTTPException(status, msg)`, if desired.

# All API Methods with FastAPI (2 of 2)

```

@app.put('/api/celebrities/{celeb_id}')
def modify_celeb(celeb_id: int, celeb: Celebrity):
    celeb.celeb_id = celeb_id
    matches = [c for c in celebs if c.celeb_id == celeb.celeb_id]
    if not matches:
        raise HTTPException(404, detail='No matching items.')
    index = celebs.index(matches[0])
    celebs[index] = celeb
    return celebs[index]

@app.delete('/api/celebrities/{celeb_id}')
def remove_celeb(celeb_id: int):
    matches = [celeb for celeb in celebs if celeb_id == celeb.celeb_id]
    if not matches:
        raise HTTPException(404, detail='No matching items.')
    celebs.remove(matches[0])
    return matches[0]

@app.exception_handler(RequestValidationError)
async def exception_handler(req: Request, exc: RequestValidationError):
    root_error_info = exc.errors()[0]
    return JSONResponse(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        content={ 'error': 'Your object contained invalid values.',
                  'info': root_error_info.get('msg')},
    )
uvicorn.run(app, host='127.0.0.1', port=8000)

```

ch06\_apdx\_b/03\_fastapi\_celeb\_full.py

A custom exception handler is shown, though not actually necessary in this case because a default handler already exists to respond to Pydantic ValidationErrors. The handler above responds to any validation errors that occur on the Pydantic models and responds by returning a 422 custom JSON response.

The PUT method finds the desired celebrity by looking up its id, then replaces the celebrity in the list with the one passed in. The DELETE method finds the celebrity in the list by its id, then removes it from the list.

## WSGI vs ASGI

- **WSGI** is a Python specification that dictates to server vendors (framework creators) the expected interface of the server
  - It was created in 2003
  - It is analogous to the Java Servlet API, but for Python
- **ASGI** (Asynchronous Server Gateway Interface) is a newer Python specification that is similar to WSGI but for asynchronous solutions
  - To be ASGI compliant the server must be build using Python's **asyncio** module

For more on ASGI, see <https://asgi.readthedocs.io/en/latest/>.

# Asyncio in a Nutshell

```

import asyncio
from itertools import islice

async def process_data(filepath: Path):
    with filepath.open(encoding='utf-8') as f:
        for line in islice(f, 10):
            await asyncio.sleep(0) ← This line allows control to return to the event loop, thus files are processed concurrently
            print(f'{filepath.name}: {line.strip()}')

async def main(files: list[str]):
    files = (Path(f'../data/{filename}') for filename in files)
    tasks = (process_data(f) for f in files)
    await asyncio.gather(*tasks)

data_files = ['car_crashes.csv', 'accounts.csv', 'contacts.dat']
asyncio.run(main(data_files))

```

Files are read concurrently

car\_crashes.csv: 18.8,7.332000000000001,5.64,18.048000000000002,  
accounts.csv: 100,John Smith,5500,0.025,C  
contacts.dat: Sally Smith,220 E. Main Ave Phila. PA 09119, ...  
car\_crashes.csv: 18.1,7.421,4.525,16.290000000000003,17.014, ...  
accounts.csv: 101,Sally Jones,6710.11,0.025,C  
contacts.dat: John Brown,231 Oak Blvd. Black Hills SD 82101, ...

ch06\_apdx\_b/05\_await.py

## Gunicorn and Uvicorn

- **Flask** was designed with Python's limitations
  - Only a single-core in a CPU can be utilized at a time in any Python process
- **Gunicorn** creates workers to handle separate requests
  - Workers are separate processes that enable parallelism (i.e., separate CPU cores can be enabled)
- **Uvicorn** uses an event loop framework called **uvloop**
  - Uvloop is an asyncio solution (though it isn't the asyncio event loop, just a different event loop)
  - It adheres to the ASGI spec, other ASGI servers include: Hypercorn and Daphne
  - FastAPI was built for asynchronous performance

# Introduction to API Development Using Python

Exercise Workbook

# Task 1-1

## Setting Up Your Environment



### Overview

This task establishes your development environment by setting up a virtual environment, and IDE, and the Python interpreter.



### Determine Your Python Command

Every Python environment is just a little different from the next. To perform this task, you need to know which Python is the correct one to use for this course. Open a command window (Windows) or a terminal (Linux/OS X). Find the correct version of Python you intend to use by typing in the following commands:

```
python -v
```

```
python3 -v
```

```
python3.10 -v      (repeat this using 3.9, 3.8, 3.7, or 3.6 as needed)
```

One of these commands should work for you. If not, you will need to check your PATH environment variable and double-check the location that Python was installed.

***Remember the correct python command that worked for you as it will be used throughout the course!***

In this course, replace any Python command-line references (e.g., python, python3, etc.) with the version you just determined works for you!



## Create and Activate a Virtual Environment

In this step we will create a virtual environment to work from. This way, anything we install will only go into the virtual environment and won't affect your current Python environments.

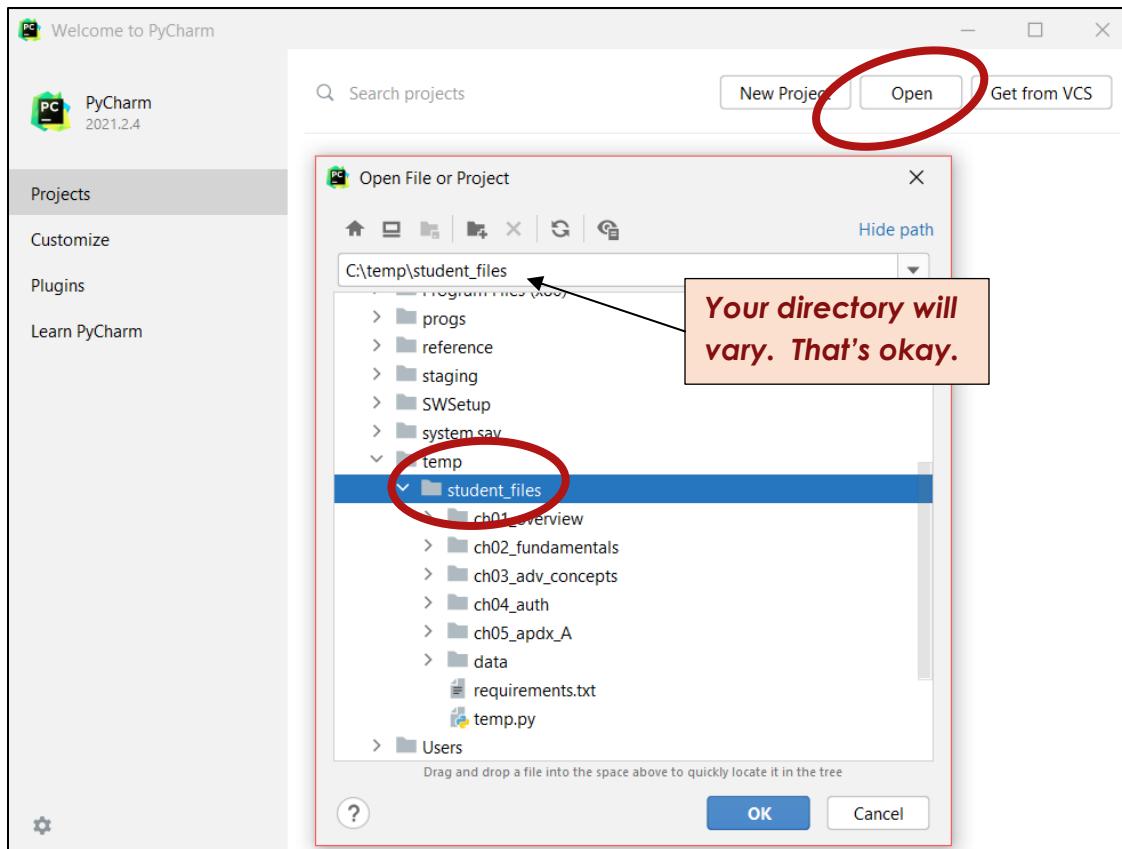
There are two ways to create the virtual environment:

1. Via PyCharm (easier, and our preferred way)
2. Manually from the command line

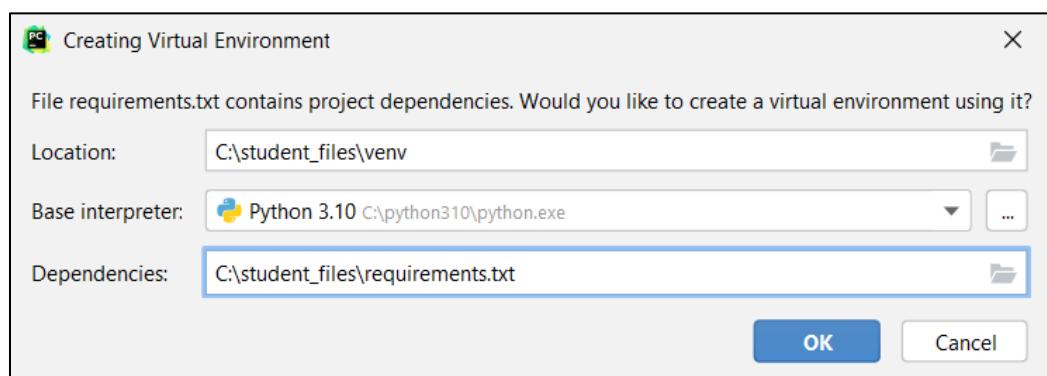
If technique #1 from above doesn't work, skip down to the star shown below and perform a manual creation of the virtual environment.

### ***Creating the Virtual Environment using PyCharm***

Launch PyCharm. In the Welcome dialog, select “Open” and browse to your student\_files directory (which should be extracted from the provided zip file already).



PyCharm should offer to create a virtual environment for you AND should offer to run the file called **requirements.txt**. Use the defaults that are provided and **click Ok**--you don't need to match the paths shown below.



That should be it for this step.

Note: If PyCharm doesn't ask to create a virtual environment for you automatically, double-click the *requirements.txt* file within your *student\_files* folder and choose the option to install dependencies mentioned in the yellow bar that should appear at the top of the editor. If no yellow bar appears, you may have to manually run the *requirements.txt* file from a Terminal within PyCharm. Ask the instructor for help on how to do this.



### **Alternatively, Create the Virtual Environment Manually**

If PyCharm isn't used to create the virtual environment for you, or PyCharm failed to do this properly, follow this next section instead.

In the same terminal window from step 1, create a directory for your virtual environment. You may create this directory anywhere. A nice possible location for this virtual environment is in a *virtualenv* directory in your home. If you don't have a location to place virtual environments on your system, type the following:

On Windows:      **cd /D %userprofile%**

On Linux/OS X:    **cd ~**

Create a directory here called **virtualenvs** by typing:

**mkdir virtualenvs**

Change into this new directory.

**cd virtualenvs**

Now create the Python virtual environment and activate it by typing:

```
python -m venv api_env
```

On Windows:      **cd api\_env\Scripts**  
                     **.\activate**                          (or just **activate**)

On Linux/OS X:    **cd api\_env/bin**  
                     **source ./activate**

Verify that you have activated the python environment by typing:

On Windows:      **where python**

On Linux/OS X:    **which python**

Check that the response shows your new virtual Python interpreter listed first.

We are about to install any needed dependencies for the course. Before doing that, you will need to ensure that the student files have been placed on your machine (already extracted from the provided zip file). Take note of the location of your student files. We'll refer to this location as your <student\_files\_home>.

```
cd <student_files_home> (this should be your location of the student files)
```

Install the dependencies:

(from the student\_files directory)

```
pip install -r requirements.txt
```

Below is a sample screenshot (with slightly different directories used) of what to expect when running this command:

```
(api_env) >pip -V
pip 21.2.4 from c:\virtualenvs\api_env\lib\site-packages\pip (python 3.10)

(api_env) c:\student_files>pip install -r requirements.txt
Collecting Flask==2.0.2
  Using cached Flask-2.0.2-py3-none-any.whl (95 kB)
Collecting SQLAlchemy==1.4.29
  Using cached SQLAlchemy-1.4.29-cp310-cp310-win_amd64.whl (1.5 MB)
Collecting requests==2.27.1
  Using cached requests-2.27.1-py2.py3-none-any.whl (63 kB)
Collecting flask-restx==0.5.1
  Using cached flask_restx-0.5.1-py2.py3-none-any.whl (5.3 MB)
Collecting Flask-sqlalchemy==2.5.1
  Using cached Flask_SQLAlchemy-2.5.1-py2.py3-none-any.whl (17 kB)
Collecting flask-marshmallow==0.14.0
  Using cached flask_marshmallow-0.14.0-py2.py3-none-any.whl (10 kB)
Collecting marshmallow-sqlalchemy==0.27.0
  Using cached marshmallow_sqlalchemy-0.27.0-py2.py3-none-any.whl (15 kB)
Collecting beautifulsoup4==4.10.0
  Using cached beautifulsoup4-4.10.0-py3-none-any.whl (97 kB)
Collecting prettytable
  Using cached prettytable-3.2.0-py3-none-any.whl (26 kB)
Collecting click>=7.1.2
  Downloading click-8.0.4-py3-none-any.whl (97 kB)
    |██████████| 97 kB 420 kB/s
Collecting itsdangerous>=2.0
  Downloading itsdangerous-2.1.2-py3-none-any.whl (15 kB)
```

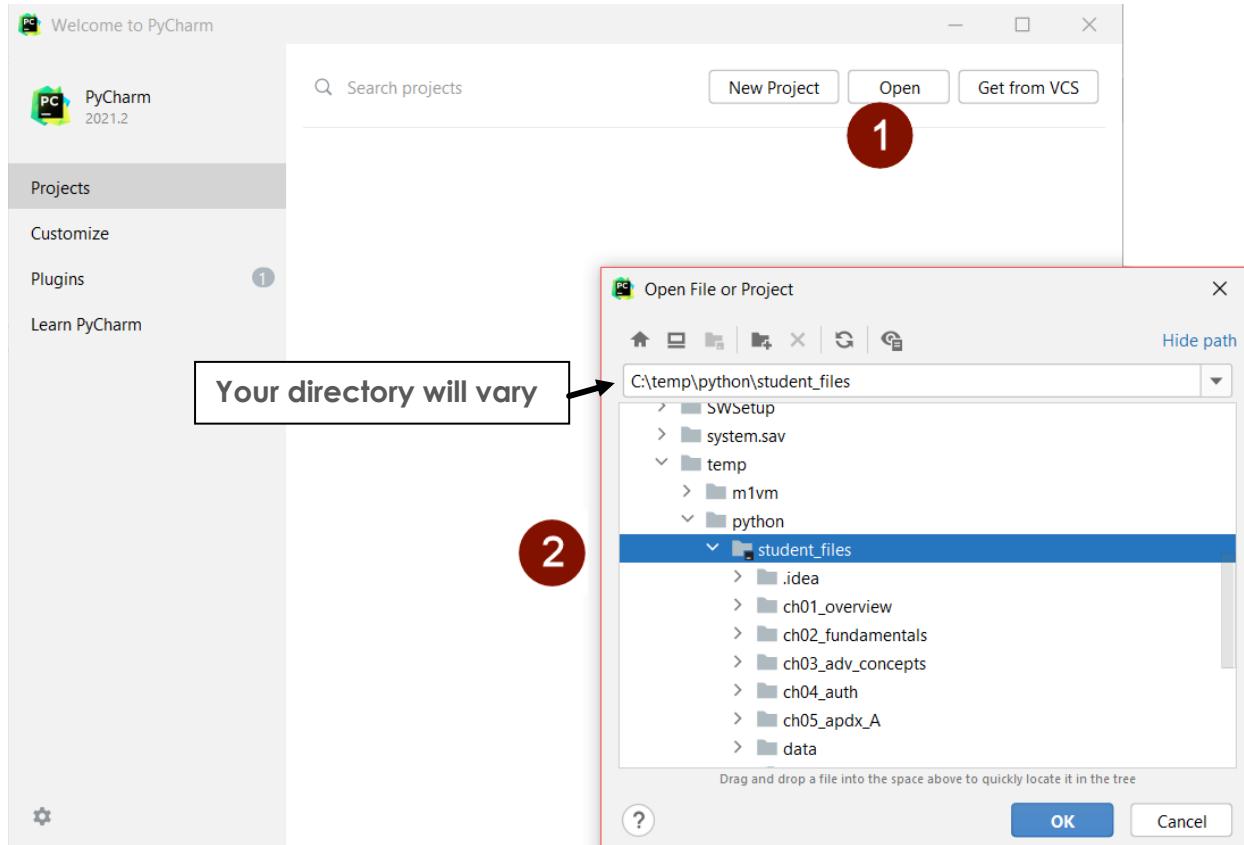


## Setup the IDE

*Note: This step may be skipped entirely if PyCharm automatically installed the packages for you when you launched the IDE in step 2.*

After the packages finish installing (should be relatively quick), **launch PyCharm** if you haven't already.

In the Welcome screen, locate the student\_files directory, select this directory and then click **Open**. In some cases, the project may open directly skipping this step. That is okay.

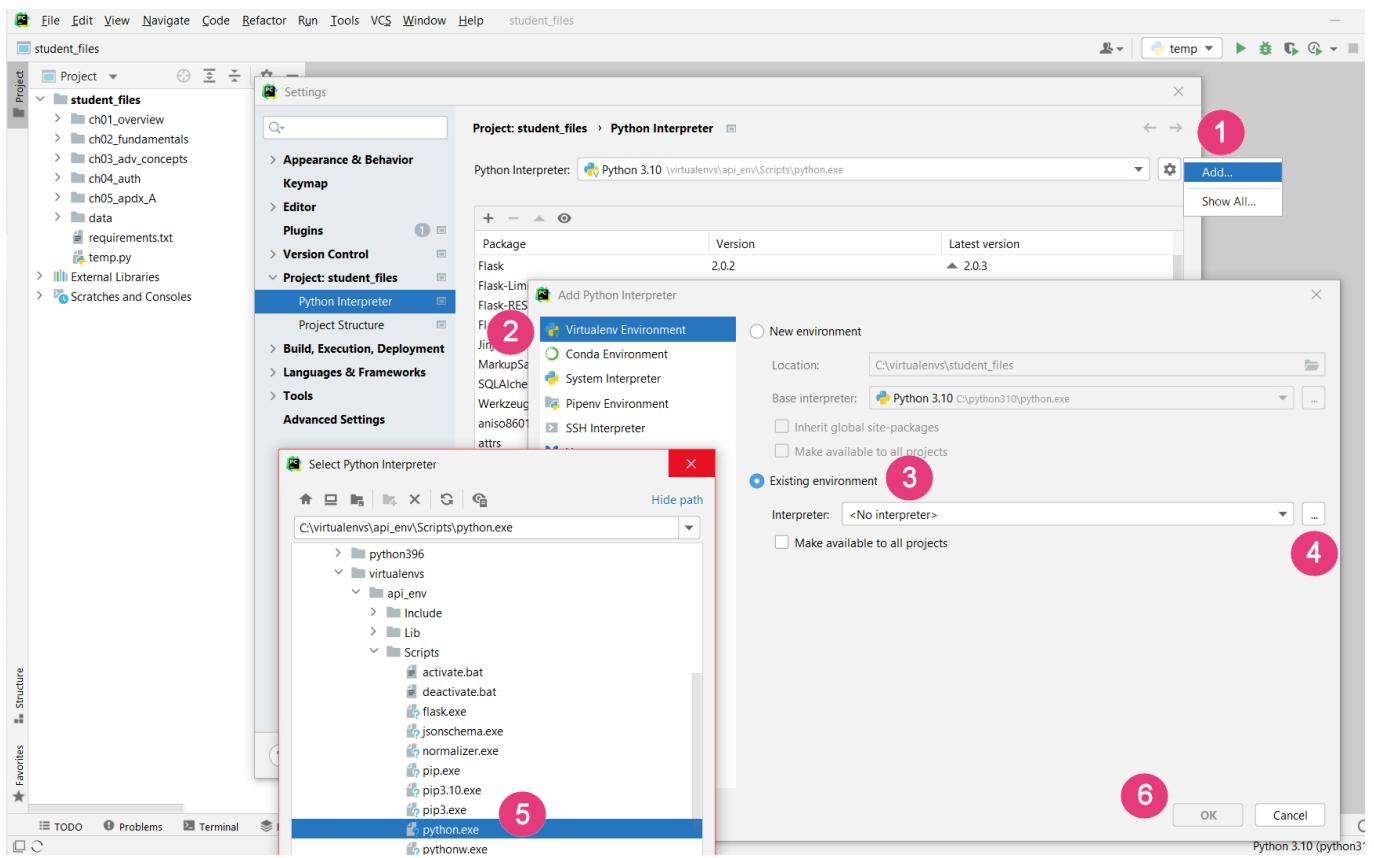


Proceed to the settings to establish which interpreter to use.

OS X: **PyCharm > Preferences**

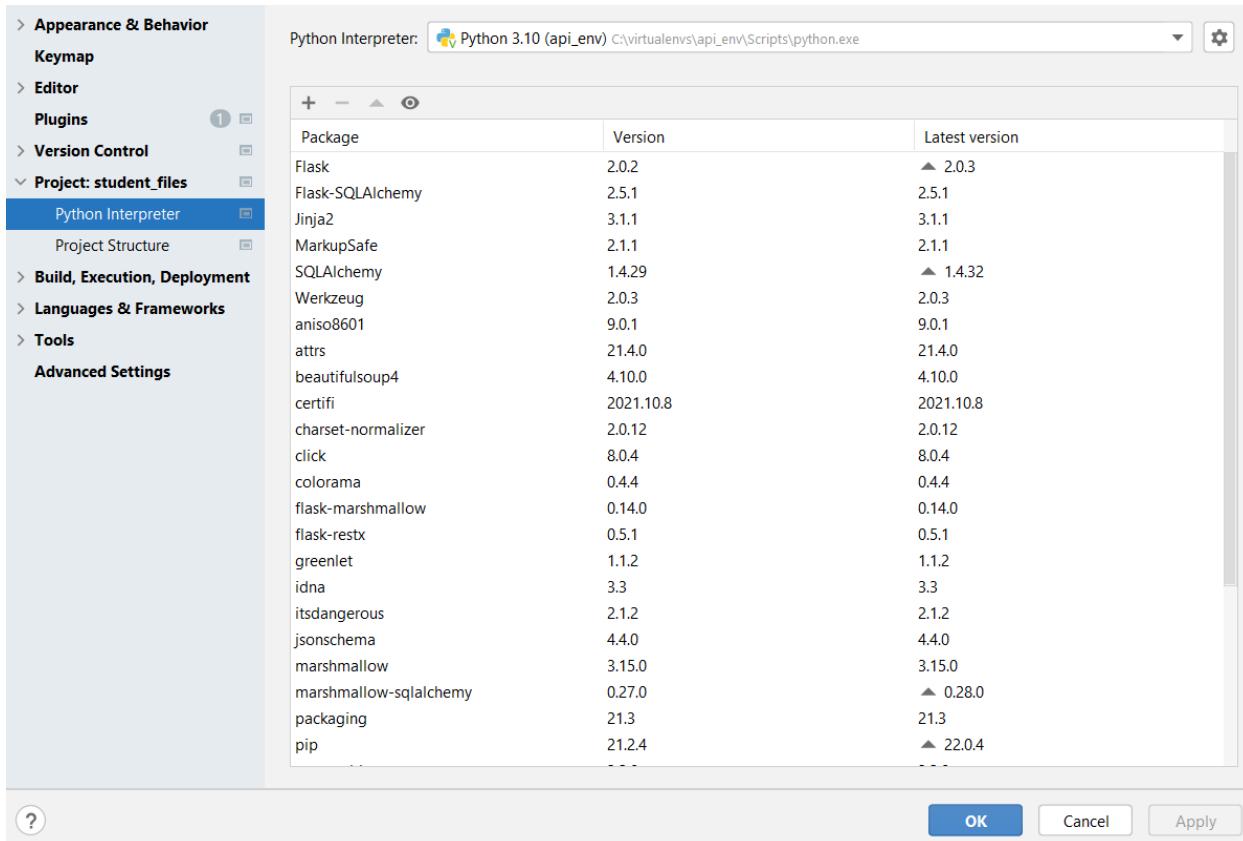
Windows: **File > Settings**

Expand the **Project: student\_files** item. Select **Project Interpreter**. On the right side, we'll have to point PyCharm to the newly created virtual environment. Do this as shown below:



- 1- Select the setting (cog wheel) symbol
- 2- Leave selected "Virtualenv Environment" option
- 3- Select "Existing Environment" button
- 4- Select the ... button (far to the right)
- 5- Browse to the new Interpreter location (python.exe on Windows, python or python3.x on OS X)
- 6- Click OK twice.

You should see numerous packages listed here now (See next screen shot)



Test out the new environment by running the file in ch01\_overview called **task1\_1\_starter.py**. To run it, right-click in the source code and select **Run task1\_1\_starter**.

That's it! If it runs, you're done!

# Task 1-3

## *Creating a Client Using Requests*



### Overview

This simple task will utilize the Python requests module to make a request and view the results of the URLs encountered in Task1-2.

Work from the provided starter file, task1\_3\_starter.py, found in the ch01\_overview directory.



### Create a for-loop. Retrieve the Data.

Create a for-loop to iterate over the URLs. Make a request to the specific URL using the requests module.

```
for url in urls:  
    r = requests.get(url)
```



### Display the Responses

Display the results from the request using

```
for url in urls:  
    r = requests.get(url)  
  
    print(r.url)  
    print('-'*len(r.url))  
    print(r.text)
```

That's it! Test it out!

# Task 2-1

## An Initial API



### Overview

This task begins the development of our invoicing API. For this first version, you will create the code to initialize the Flask server and set up our first route. Begin by opening the task2\_1\_starter.py file within ch02\_fundamentals.



### Create the Flask Server

At the location of step 1 in the source file, instantiate the Flask object passing the name of our module into it.

```
app = Flask(__name__)
```



### Define a Route

Define a single route for now. It will only handle "GET" HTTP requests. It will map to: **/api/invoices/<invoice\_num>**. Place this route above the retrieve\_invoice() function.

```
@app.route('/api/invoices/<invoice_num>', methods=['GET'])
def retrieve_invoice(invoice_num):
```



## Retrieve a Matching Invoice Number

Within the `retrieve_invoice()` function, write a list comprehension that builds a list (called `results`) by iterating through the list of invoices (called `data`) and checking for any invoice numbers (`invoice_num`) that match the invoice id (`row[0]`).

```
results = [row[1:] for row in data if invoice_num == row[0]]
```



## JSONify a Response

Pass the results back in a `jsonify` call along with the invoice number. Use `results=` and `invoice=` as the keys in the `jsonify()` call.

```
resp = jsonify(results=results, invoice=invoice_num)
```

That's it! Test it out within your browser for now by starting the server and navigating your browser to: <http://localhost:8051/api/invoices/536365>.

# Task 2-2

## An API Client



### Overview

This task will allow us to interact with our invoice API via a Python-based client. We'll work from the provided task2\_2\_starter.py file.



### Prompt the User for an Invoice Number

Use a simple prompt to ask the user to supply an invoice number. There are many ways to retrieve an input from the user. Below represents only one possible way. Feel free to create your own, otherwise, use the technique shown below:

```
default = '536365'  
invoice_num = input('Enter invoice number (def. 536365): ')  
if not invoice_num:  
    invoice_num = default
```



### Construct the URL to Access Our Flask Server

We need to combine the provided base\_url, path, and user's input to create the URL. A Python f-string can do this.

```
default = '536365'  
invoice_num = input('Enter invoice number (def. 536365): ')  
if not invoice_num:  
    invoice_num = default  
  
url = f'{base_url}{path}{invoice_num}'
```



## Make a Request and Convert the Response

Time to make an HTTP GET request using the requests module and the URL created in step 2. Convert the returned response to JSON using the .json() method:

```
default = '536365'  
invoice_num = input('Enter invoice number (def. 536365): ')  
if not invoice_num:  
    invoice_num = default  
  
url = f'{base_url}{path}{invoice_num}'  
  
results = requests.get(url).json()
```



## Display the Retrieved Results

Use json.dumps() to take the returned dictionary and output it as a string.

```
default = '536365'  
invoice_num = input('Enter invoice number (def. 536365): ')  
if not invoice_num:  
    invoice_num = default  
  
url = f'{base_url}{path}{invoice_num}'  
  
results = requests.get(url).json()  
print(json.dumps(results, indent=4))
```

That's it! Test it out by:

1. Making sure you are running the task2\_1\_starter.py server.
2. Running the task2\_2\_starter.py file after the server is already running.

# Task 2-3

## Using Flask-RESTX



### Overview

This task will implement the Flask-RESTX plugin (that should already be installed from Task1-2). You will create two classes: Invoice and Invoices. You will implement several RESTful methods within them. You will run the server and finally test out your solution using the provided task2\_3\_client.py file.



### Import Needed Items

Add the imports needed to work with the Flask-RESTX plugin. These are **Resource** and **API**.

```
from flask import Flask, jsonify, Response
from flask_restx import Resource, Api
```



### Create the Primary Flask-RESTX Object

Create the `Api()` object. You can also specify a prefix that will be expected within all URLs. We'll use `"/api"` for the prefix:

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

app = Flask(__name__)
api = Api(app, prefix='/api')
```



## Create Invoice and Invoices

Create the two classes with pass statements for now. The two classes are Invoice and Invoices. They should each inherit from Flask-RESTX's Resource class.

```
data_file = '.../.../data/customer_purchases.csv'  
data = [line.strip().split(',') for line in  
       Path(data_file).open(encoding='unicode_escape')][1:]  
print('Customer purchase data read.')
```

```
class Invoices(Resource):  
    pass
```

```
class Invoice(Resource):  
    pass
```



## Register the Classes with Flask-RESTX

After the class definitions, add two lines to register our classes with Flask-RESTX. This way we won't have to do it with a decorator.

```
class Invoice(Resource):  
    pass  
  
api.add_resource(Invoice, '/invoices/<invoice_num>')  
api.add_resource(Invoices, '/invoices/')
```



## Refactor the retrieve\_invoice() Method

Create a get() method in the Invoice class. Code created in Task 2-1 in the retrieve\_invoice() method can be moved into the Invoice class get() method. Be careful not to put it in the Invoices class get() method. The jsonify() and Response lines of code do not need to transfer. Remove the decorator. Remove the retrieve\_invoice() method once the get() is complete

```
class Invoice(Resource):
    def get(self, invoice_num):
        results = [row[1:] for row in data
                   if invoice_num == row[0]]
```



## Create get() and post() within Invoices

Create get() and post() methods within Invoices. For now, remove the pass statement from the class and add pass into each of the two new methods.

```
class Invoices(Resource):
    def get(self):
        pass

    def post(self):
        pass
```



## Create put() and delete() within Invoice

In the Invoice class, add a put() and delete() method that each take an invoice\_num parameter after self. Have each return pass for now.

```
class Invoice(Resource):
    def get(self, invoice_num):
        results = [row[1:] for row in data
                   if invoice_num == row[0]]

    def put(self, invoice_num):
        pass

    def delete(self, invoice_num):
        pass
```



## Return Temporary Sample Responses

In the post(), put(), and delete() methods, remove the pass statements, have these methods return temporary objects as shown. In the Invoices get(), have it return the first 99 invoices. Have the Invoice get() method return the results data.

```
class Invoices(Resource):
    def get(self):
        return {'results100': data[1:100]}

    def post(self):
        return {'action': 'post'}
```

```
class Invoice(Resource):
    def get(self, invoice_num):
        results = [row[1:] for row in data if invoice_num ==
row[0]]
        return {'results': results}

    def put(self, invoice_num):
        return {'action': 'put'}

    def delete(self, invoice_num):
        return {'action': 'delete'}
```

That's it! Test it out by:

1. Ensuring no other servers are running. Be sure to stop them.
2. Starting the task2\_3\_starter.py server.
3. Using the provided client (task2\_3\_client.py) to access the server.

# Task 2-4

## Using Postman (Optional)



### Overview

This task will use (or demonstrate the use of) Postman. If you don't have Postman installed on your working machine, this is okay. The exercise provides screen shots to demonstrate its capabilities.

You will need Postman desktop for this task as it is required for localhost testing. Open Postman desktop app if you have it.



### Create the Invoice Collection

Open the Postman desktop app. In the **My Workspace** section on the left, select **Collections**. Then, in the field to the right, define the name **Invoice API**. (See the screen shot).

The screenshot shows the Postman application interface. On the left, there's a sidebar titled 'My Workspace' with options like 'Collections', 'APIs', 'Environments', etc. A red circle highlights the 'Collections' icon. On the right, a main panel shows a collection named 'Invoice API'. Another red circle highlights the collection name 'Invoice API'. Below it, tabs for 'Authorization', 'Pre-request Script', 'Tests', and 'Variables' are visible. A note says 'This authorization method will be used for every request in this collection. You can'. Under the 'Type' section, it says 'No Auth'. A note below states 'This collection does not use any authorization. Learn more about authorization'.

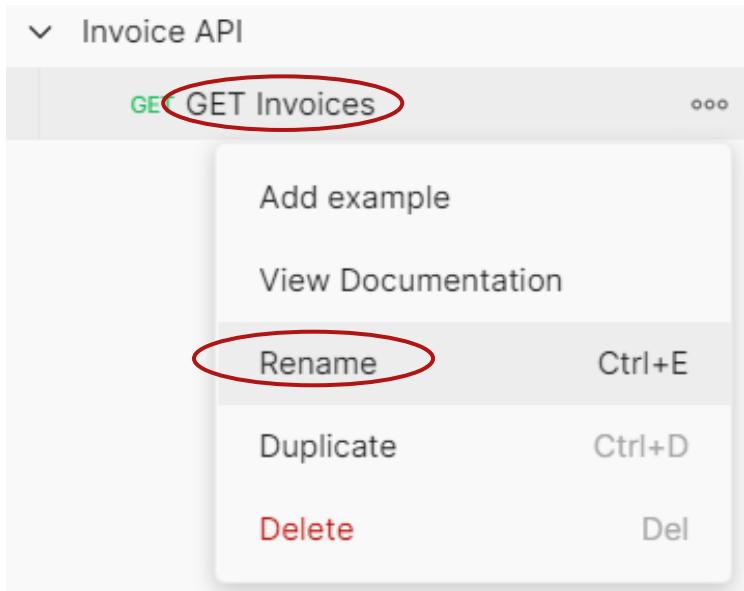


## Add the Five Requests

Next, add a request to the collection as shown below.

The screenshot shows a context menu for the 'Invoice API' collection. The 'Add Request' option is highlighted with a red circle. Other options include 'Add Folder', 'Monitor Collection', 'Mock Collection', 'Create a fork', 'Rename' (with keyboard shortcut Ctrl+E), 'Duplicate' (with keyboard shortcut Ctrl+D), 'Export', and 'Delete'.

Give the request a new name.



Supply the URL of `http://localhost:8051/api/invoices`.

Select the HTTP method type as shown.

A screenshot of the Postman application. On the left, there's a sidebar with a tree view of APIs and requests. Under 'Invoice API', the 'GET GET Invoices' request is selected and circled in red. The main panel shows the 'Invoice API / GET Invoices' tab. At the top of this tab, the method 'GET' is circled in red, and the URL 'http://localhost:8051/api/invoices' is also circled in red. Below the URL, there are tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. Under the 'Headers' tab, there's a section titled 'Query Params' with a table. The table has columns for 'KEY' and 'VALUE'. There is one row in the table with the value 'Key' under 'KEY' and 'Value' under 'VALUE'.

Save the tab (request) using CTRL-S.

Do this again this time for the **POST** request next:

1. Create a new request.
2. Rename it.
3. Use the same URL from the previous step.
4. Be sure to select POST type.
5. Save the tab (request) using CTRL-S.

The screenshot shows the Postman interface with a request named "POST POST Invoices". The request is of type "POST" and points to the URL "http://localhost:8051/api/invoices". The "Body" tab is selected, and the "form-data" radio button is chosen. Below the table, there is a "Response" section containing two small icons: a rocket ship and a target.

| KEY          | VALUE                  | DESCRIPTION |
|--------------|------------------------|-------------|
| invoice_no   | 581588                 |             |
| stock_code   | 20961                  |             |
| quantity     | 3                      |             |
| description  | STRAWBERRY BATH SPONGE |             |
| invoice_date | 2/2/2022               |             |
| unit_price   | 2.46                   |             |
| customer_id  | 17850                  |             |
| country      | United Kingdom         |             |
| Key          | Value                  | Description |

Add parameters by selecting the Body tab and then the form-data radio button as shown above. Add in values: **invoice\_no**: 581588, **stock\_code**: 20961, **quantity**: 3, **description**: STRAWBERRY BATH SPONGE, **invoice\_date**: 2/2/2022, **unit\_price**: 2.46, **customer\_id**: 17850, **country**: United Kingdom.

Save the request again.

Now for the GET (singular) request:

1. Create a new request.
2. Rename it.
3. Select a specific invoice to test (see screen shot). We used **536365**.

The screenshot shows the Postman application interface. On the left, there's a sidebar with a '+' button, a filter icon, and a 'Collection' dropdown. Below that, under 'Celebrity API', is a 'GET Celebrities' request. Under 'Invoice API', which is expanded, are five requests: 'GET Invoices' (POST), 'POST POST Invoices' (GET), 'GET GET Invoice' (selected and highlighted in grey), 'PUT PUT Invoice' (PUT), and 'DEL DELETE Invoice' (DELETE). The main panel title is 'Invoice API / GET Invoice'. At the top right, the method 'GET' is circled in red, and the URL 'http://localhost:8051/api/invoices/536365' is also circled in red. Below the URL, tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', and 'Tests' are visible, with 'Params' being the active tab. A table titled 'Query Params' has two columns: 'KEY' and 'VALUE'. A single row is present with 'Key' in the KEY column and 'Value' in the VALUE column. At the bottom of the main panel, the word 'Response' is centered.

Be sure to save the tab (request) with CTRL-S.

Implement the PUT and DELETE requests (as shown below) using the same technique as the GET (singular) shown above. Don't forget to save the new requests each time. For the PUT request, add the form parameters into the Body > Form Data section as we did for the PUT request.

The screenshot shows the Postman interface with the following details:

- Left Sidebar:** Shows the project structure with "Celebrity API" expanded and "Invoice API" selected.
- Request URL:** `http://localhost:8051/api/invoices/536365`
- Method:** **PUT** (highlighted with a red circle)
- Tab Selection:** Headers (8) (highlighted with a red underline)
- Body Tab:** Contains a table for "Query Params" with columns **KEY** and **VALUE**. It has one row with "Key" and "Value".
- Context Menu (Open in Tab):** Shows options: Open in Tab, Rename (Ctrl+E), Duplicate (Ctrl+D), Delete (Del).

Again, for the PUT, add Body parameters as we did for the POST request (not pictured this time).

Add parameters by selecting the Body tab and then the form-data radio button as shown above. Add in values: **invoice\_no**: 581588, **stock\_code**: 20961, **quantity**: 3, **description**: STRAWBERRY BATH SPONGE, **invoice\_date**: 2/2/2022, **unit\_price**: 2.46, **customer\_id**: 17850, **country**: United Kingdom.

Save the request again.

Create the DELETE request. No body parameters are needed in this case.

The screenshot shows the Postman interface with the following details:

- Left Sidebar:** Shows the project structure with "Celebrity API" expanded and "Invoice API" selected.
- Request URL:** `http://localhost:8051/api/invoices/536365`
- Method:** **DELETE** (highlighted with a red circle)
- Tab Selection:** Headers (7) (highlighted with a red underline)
- Body Tab:** Contains a table for "Query Params" with columns **KEY** and **VALUE**. It has one row with "Key" and "Value".
- Context Menu (Open in Tab):** Shows options: Open in Tab, Rename (Ctrl+E), Duplicate (Ctrl+D), Delete (Del).

Test it out by running your task2\_3\_starter.py server. Then, for each request in the collection, select it. Press send and view the output results, as shown:

The screenshot shows the Postman interface with the 'Invoice API / GET Invoices' collection selected. The 'GET Invoices' request is highlighted with a red circle. The 'Send' button at the top right is also circled in red. Below the request, the 'Raw' tab is selected in the preview area, showing the JSON response data.

```
{"results100": [[{"id": "536365", "product": "71053", "name": "WHITE METAL LANTERN", "category": "6", "date": "12/1/2010 8:26", "price": "3.39", "country": "United Kingdom"}, {"id": "536365", "product": "84466B", "name": "CREAM CUPID HEARTS COAT HANGER", "category": "8", "date": "12/1/2010 8:26", "price": "2.75", "country": "United Kingdom"}, {"id": "536365", "product": "84029G", "name": "KNITTED UNION FLAG HOT WATER BOTTLE", "category": "6", "date": "12/1/2010 8:26", "price": "3.39", "country": "United Kingdom"}, {"id": "536365", "product": "84029E", "name": "RED WOOLLY HOTTIE WHITE HEART.", "category": "6", "date": "12/1/2010 8:26", "price": "3.39", "country": "United Kingdom"}, {"id": "536365", "product": "22752", "name": "SET 7 BABUSHKA NESTING BOXES", "category": "2", "date": "12/1/2010 8:26", "price": "7.65", "country": "United Kingdom"}, {"id": "536365", "product": "21730", "name": "GLASS STAR FROSTED T-LIGHT HOLDER", "category": "6", "date": "12/1/2010 8:26", "price": "4.25", "country": "United Kingdom"}, {"id": "536366", "product": "226331", "name": "HAND WARMER UNION JACK", "category": "6", "date": "12/1/2010 8:28", "price": "1.85", "country": "United Kingdom"}, {"id": "536366", "product": "226321", "name": "HAND WARMER RED POLKA DOT", "category": "6", "date": "12/1/2010 8:28", "price": null, "country": null}]]
```

# Task 2-5

## Adding Flask-SQLAlchemy



### Overview

This task incorporates the database using a plugin called Flask-SQLAlchemy. You will add the plugin into our on-going invoice API, create a database model object, write and test the post() method.



### Add Imports

Add an import for the Flask-SQLAlchemy plugin.

```
from flask import Flask, request
from flask_restx import Resource, Api
from flask_sqlalchemy import SQLAlchemy
```



### Take Note of the Paths Used

There's no code to write for this step, just understand how the Path() objects are being used here.

```
student_files_dir = Path(__file__).parents[1]
db_file = student_files_dir / 'data/course_data.db'
if not db_file.exists():
    print(f'Database file does not exist.', file=sys.stderr)
    sys.exit()

app.config['SQLALCHEMY_DATABASE_URI'] =
    'sqlite:///{} + str(db_file)'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```



## Create (Instantiate) the SQLAlchemy Object

Pass the Flask app object into the SQLAlchemy plugin object.

```
db = SQLAlchemy(app)
print(f'Using db engine: {db.engine}')
```



## Create the Database Model Class

Build the model class. It's a bit lengthy and you can use the commented fields in the source file to shorten it. You will still have to create the `__init__()` manually.

```
class InvoiceModel(db.Model):
    __tablename__ = 'purchases'
    id = db.Column(db.Integer, primary_key=True)
    invoice_no = db.Column('InvoiceNo', db.String(30))
    stock_code = db.Column('StockCode', db.String(30))
    quantity = db.Column(db.Integer)
    description = db.Column(db.String(150))
    invoice_date = db.Column('InvoiceDate', db.String(50))
    unit_price = db.Column('UnitPrice', db.Float)
    customer_id = db.Column('CustomerID', db.String(50))
    country = db.Column(db.String(50))

    def __init__(self, invoice_no, stock_code, quantity,
                 description, invoice_date, unit_price,
                 customer_id, country):
        self.invoice_no = invoice_no
        self.stock_code = stock_code
        self.quantity = quantity
        self.description = description
        self.invoice_date = invoice_date
```

```

    self.unit_price = unit_price
    self.customer_id = customer_id
    self.country = country

def __str__(self):
    return f'{self.invoice_no} {self.stock_code} Qty:
              {self.quantity} - {self.description}'

```



## Write the post() Method

Create the post() method to insert a new object in the database. It must first extract the fields out of the request.form dictionary. Then it will use them to instantiate and insert a new database model object (InvoiceModel). For now, we will manually return a dictionary object created the hard way.

```

class Invoices(Resource):
    def get(self):
        return {'action': 'get all'}


    def post(self):
        invoice_no = request.form.get('invoice_no')
        stock_code = request.form.get('stock_code')
        quantity = int(request.form.get('quantity'))
        description = request.form.get('description')
        invoice_date = request.form.get('invoice_date')
        unit_price = float(request.form.get('unit_price'))
        customer_id = request.form.get('customer_id')
        country = request.form.get('country')

        new_purchase = InvoiceModel(invoice_no, stock_code,
                                     quantity, description, invoice_date,
                                     unit_price, customer_id, country)
        db.session.add(new_purchase)

```

```
db.session.commit()

return {'id': new_purchase.id,
        'invoice_no': new_purchase.invoice_no,
        'stock_code': new_purchase.stock_code,
        'quantity': new_purchase.quantity,
        'description': new_purchase.description,
        'invoice_date': new_purchase.invoice_date,
        'unit_price': new_purchase.unit_price,
        'customer_id': new_purchase.customer_id,
        'country': new_purchase.country }
```

That's it! Test it out by:

1. Ensuring no other servers are running
2. Starting the task2\_5\_starter.py server
3. Run the task2\_5\_client.py file to test the post operation

# Task 2-6

## Adding Flask-Marshmallow



### Overview

This last task in the chapter completes our Invoice API. In this task, you will incorporate Marshmallow, the object serialization library. You will also complete the remaining API methods in the Invoice and Invoices classes. Work from the provided task2\_6\_starter.py in the ch02\_fundamentals folder.



### Import Marshmallow

Import Marshmallow from flask-marshmallow. This should already be available for you from our Task 1-2 setup exercise. You just need to import it now.

```
from flask import Flask, request
from flask_restx import Resource, Api
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
```



### Instantiate Marshmallow

Instantiate the plugin passing the Flask object into it.

```
app = Flask(__name__)
api = Api(app, prefix='/api')
ma = Marshmallow(app)
```



## Create a Marshmallow Schema

We won't have to create the Marshmallow schema class from scratch. It is provided for us at the bottom of the source file. Locate step 3 in the source file and uncomment the Schema class and the associated lines (see below for which lines to uncomment).

```
class InvoiceSchema(ma.Schema):
    class Meta:
        fields = ('id', 'invoice_no', 'stock_code', 'quantity',
                  'description', 'invoice_date', 'unit_price',
                  'customer_id', 'country')

    invoice_schema = InvoiceSchema()
    invoices_schema = InvoiceSchema(many=True)

api.add_resource(Invoice, '/invoices/<id>')
api.add_resource(Invoices, '/invoices/')
```



## Complete the Invoices get() ALL Method

The get() method of the Invoices class will require two statements. The first queries the database, returning all records. The second returns the results from the function in a JSON format. We'll use the Marshmallow Invoices schema for that. Replace the current return statement in this function and implement these statements as shown:

```
class Invoices(Resource):
    def get(self):
        invoices = InvoiceModel.query.all()
        return {'results': invoices_schema.dump(invoices)}
```



## Complete the Invoice get() Method

Remove the current return value in this method. Complete it by using SQLAlchemy to retrieve one object by its primary key. We'll use the query.get(id) method for this. Return this object from the function using Marshmallow as shown:

```
class Invoice(Resource):
    def get(self, id):
        invoice = InvoiceModel.query.get(id)
        return invoice_schema.jsonify(invoice)
```



## Complete the Invoice put() Method

Remove the current return value. Because the put() method is a little longer, some of it has been completed for us. Since PUT performs an update, we'll first retrieve the object from the database and then update the fields to the values sent by the user (in the form). Don't forget to commit this interaction and finally return the object as JSON using Marshmallow. Do all of this as shown below:

```
def put(self, id):
    invoice_no = request.form.get('invoice_no')
    stock_code = request.form.get('stock_code')
    quantity = int(request.form.get('quantity'))
    description = request.form.get('description')
    invoice_date = request.form.get('invoice_date')
    unit_price = float(request.form.get('unit_price'))
    customer_id = request.form.get('customer_id')
    country = request.form.get('country')

    invoice = InvoiceModel.query.get(id)
    invoice.invoice_no = invoice_no
```

```
    invoice.stock_code = stock_code
    invoice.quantity = quantity
    invoice.description = description
    invoice.invoice_date = invoice_date
    invoice.unit_price = unit_price
    invoice.customer_id = customer_id
    invoice.country = country

    db.session.commit()

    return invoice_schema.jsonify(invoice)
```



## Complete the Invoice delete() Method

The delete() method will be easier than the put() method. Remove the current return value. Replace it with the code to retrieve the specific Invoice object from the database. Invoke the db.session object's delete(). Finally, return a schema object of the deleted Invoice.

```
def delete(self, id):
    invoice = InvoiceModel.query.get(id)
    db.session.delete(invoice)
    db.session.commit()
    return invoice_schema.jsonify(invoice)
```

That's it! Test it out by running task2\_6\_starter.py as a server and then running the provided client, task2\_6\_client.py.

# Task 3-1

## Pagination



### Overview

This task adds a pagination feature into the API service. To do this, you will only need to modify the Invoices get() method. We'll make use of the SQLAlchemy plugin's Paginate object. Begin by working from your task3\_1\_starter.py file in the ch03\_adv\_concepts folder.



#### Define a Page Size

In the Invoices class, add a class-level parameter to define the page size. The suggested value was 50.

```
class Invoices(Resource):
    page_size = 50
    def get(self):
```



#### Retrieve the Page Arg From the Request

Use Flask's request.args object to retrieve the submitted query string parameter. We'll use a default of 1 if nothing is submitted.

```
class Invoices(Resource):
    page_size = 50
    def get(self):
        page = request.args.get('page') or '1'
```



## Invoke the SQLAlchemy paginate() Method

Use SQLAlchemy's paginate() method, passing the page number and the page\_size. Don't forget to convert the page argument to an int.

```
class Invoices(Resource):
    page_size = 50
    def get(self):
        page = request.args.get('page') or '1'
        paged_invoices =
            InvoiceModel.query.paginate(int(page), self.page_size)
```



## Return the Paginate Object's Items

Use the returned paginate object from step 3 (we called it paged\_invoices above). Take its items attribute and pass it to Marshmallow's dump() method. Use the invoices\_schema object created previously to do this.

```
class Invoices(Resource):
    page_size = 50
    def get(self):
        page = request.args.get('page') or '1'
        paged_invoices =
            InvoiceModel.query.paginate(int(page), self.page_size)
        return
            {'results': invoices_schema.dump(paged_invoices.items)}
```

That's it! Test it out by running task3\_1\_starter.py and then running task3\_1\_client.py.