



Introduction to API Development Using Python

Participant Guide



Copyright

This subject matter contained herein is covered by a
copyright owned by: Copyright © 2024 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 / 10.14.2024

©2024 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: 10/14/2024

Notes

Chapters at a Glance

Chapter 1	Introduction and Overview	14
Chapter 2	Fundamentals	43
Chapter 3	Advanced Concepts	96
Chapter 4	Authorization and Authentication	115
	Course Summary	126
Appendix	Python Primer	131

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

Incorporating the Database

Course Agenda (*continued*)

Day 2

Completing the API

Advanced Concepts

Pagination

Versioning

Caching

Authorization and Authentication

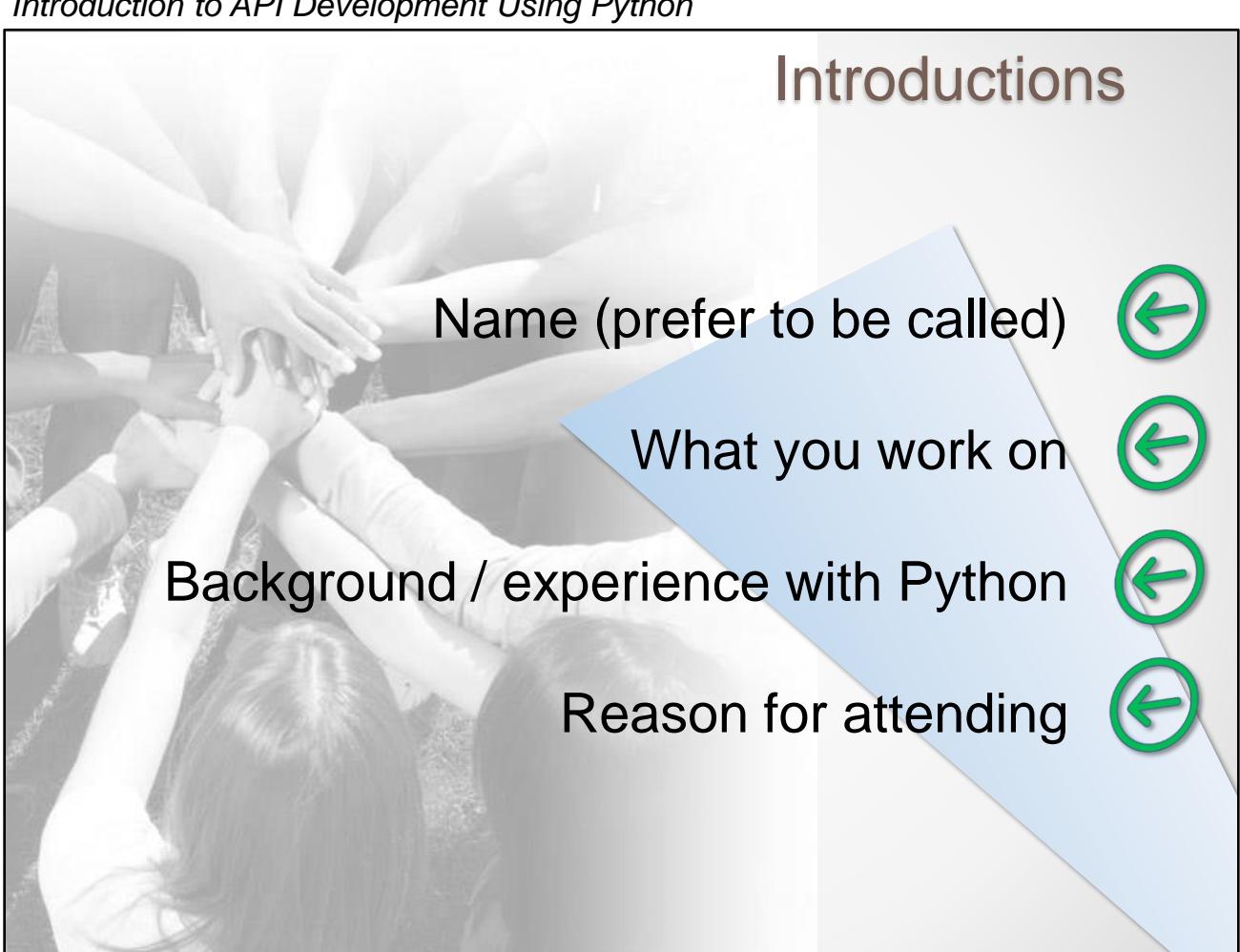
Introductions

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

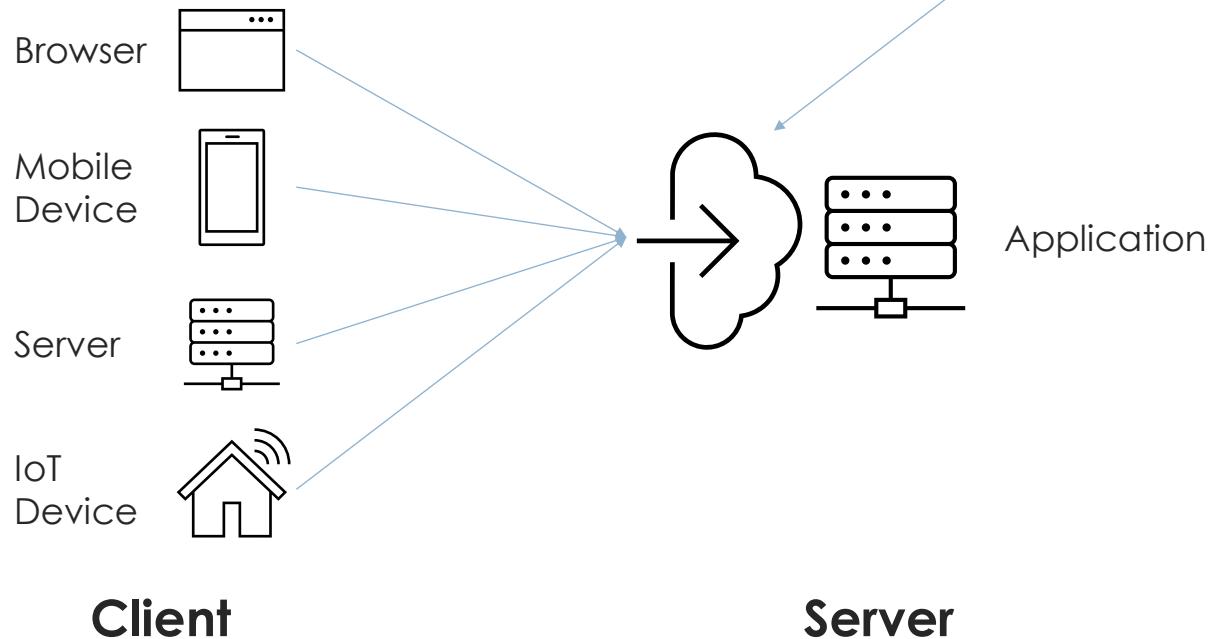
Service Architectures

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

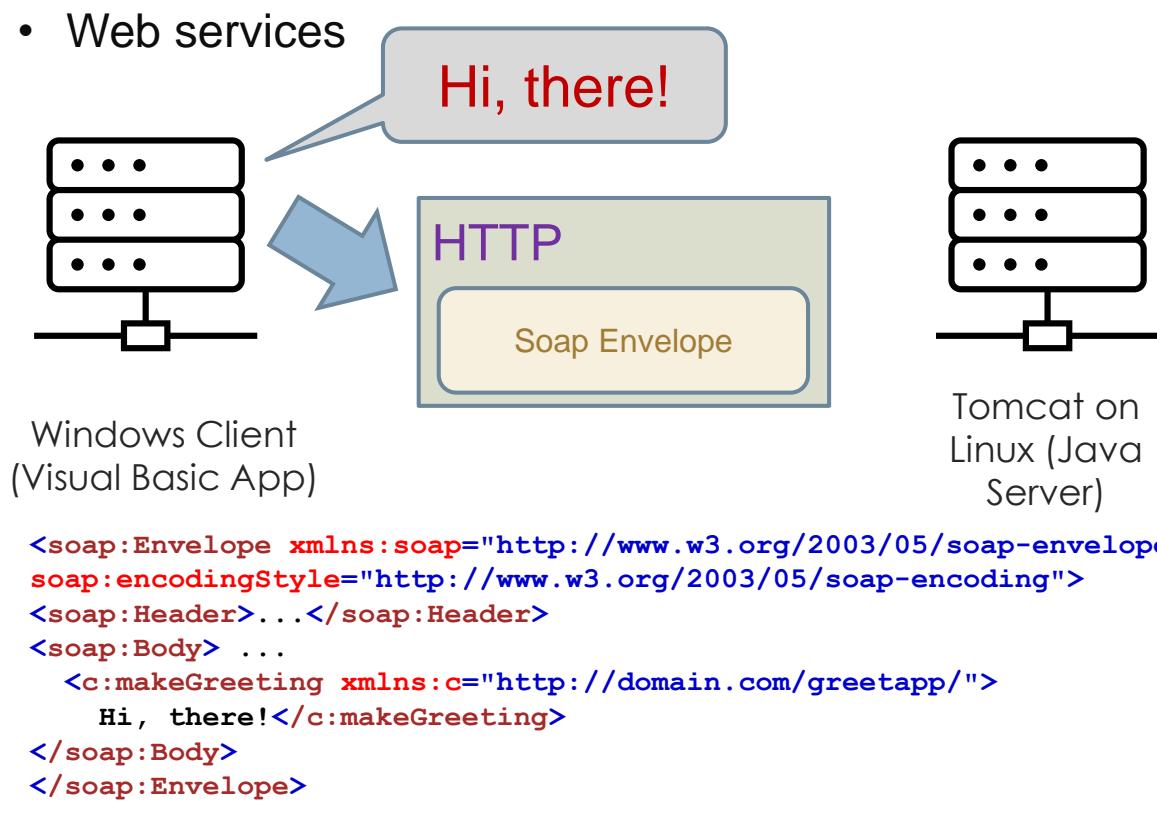
Over the years, different technologies for facilitating communication between two devices have come and gone (or largely been replaced). Many of these technologies focused on ways to transmit objects from one machine to another. They were often vendor-specific (focusing on the use of specific frameworks 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

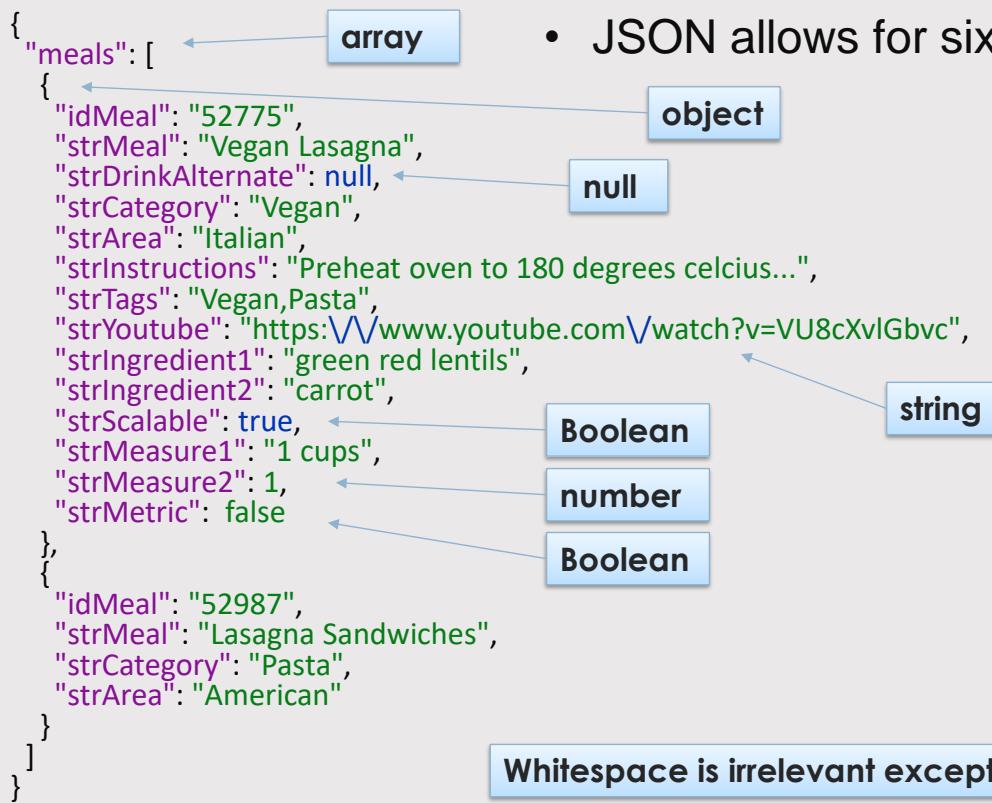
For many years, a standard for JSON did not exist. An RFC for JSON is managed by the IETF and can be found at: <https://datatracker.ietf.org/doc/html/rfc8259>.

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.

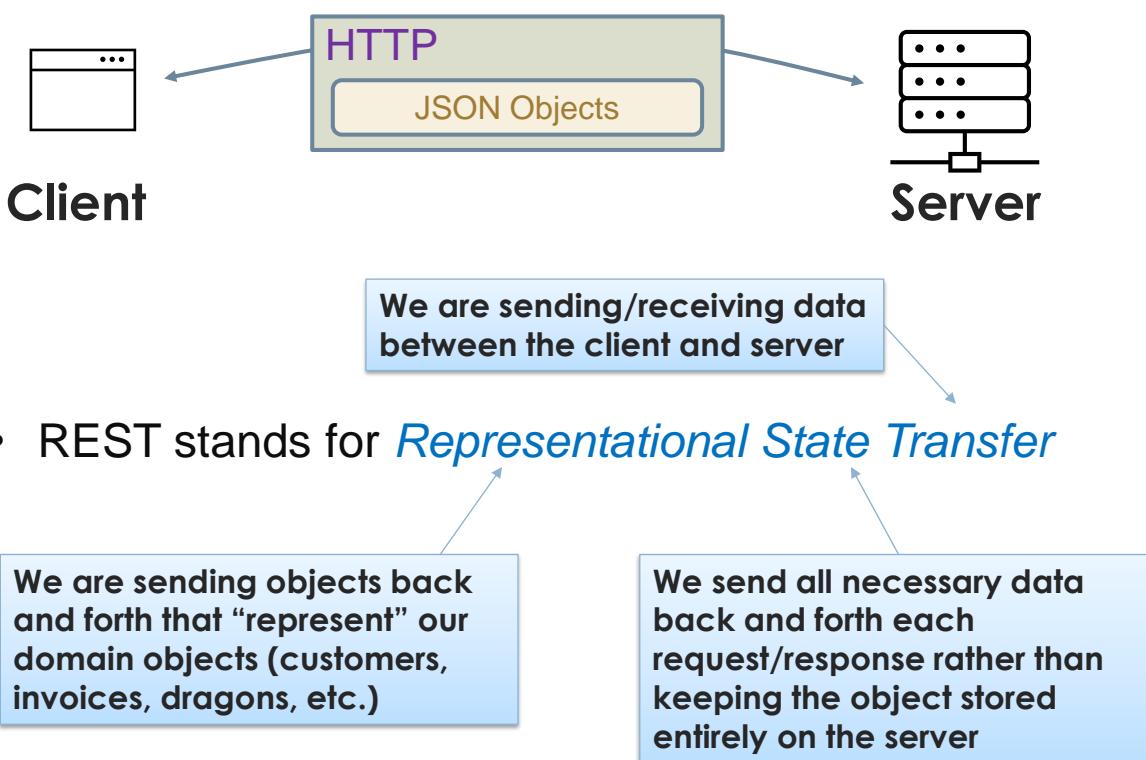
Whitespace and JSON Data



- 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 URLs
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*
 - *Orders*
 - *Patients*
 - *Customers*
 - *Documents*

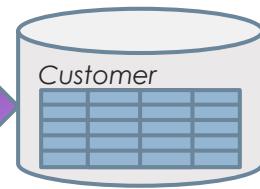
A resource can be a collection (like accounts) or singular (like account) and may contain sub-resources (like address)

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

- URIs to map to resources

`https://hostname.com/api/customers`



- Manipulation of resources via HTTP methods

GET

PUT

DELETE

POST

`https://hostname.com/api/customers/121`

`https://hostname.com/api/customers`

- Transfer of state (stateless)



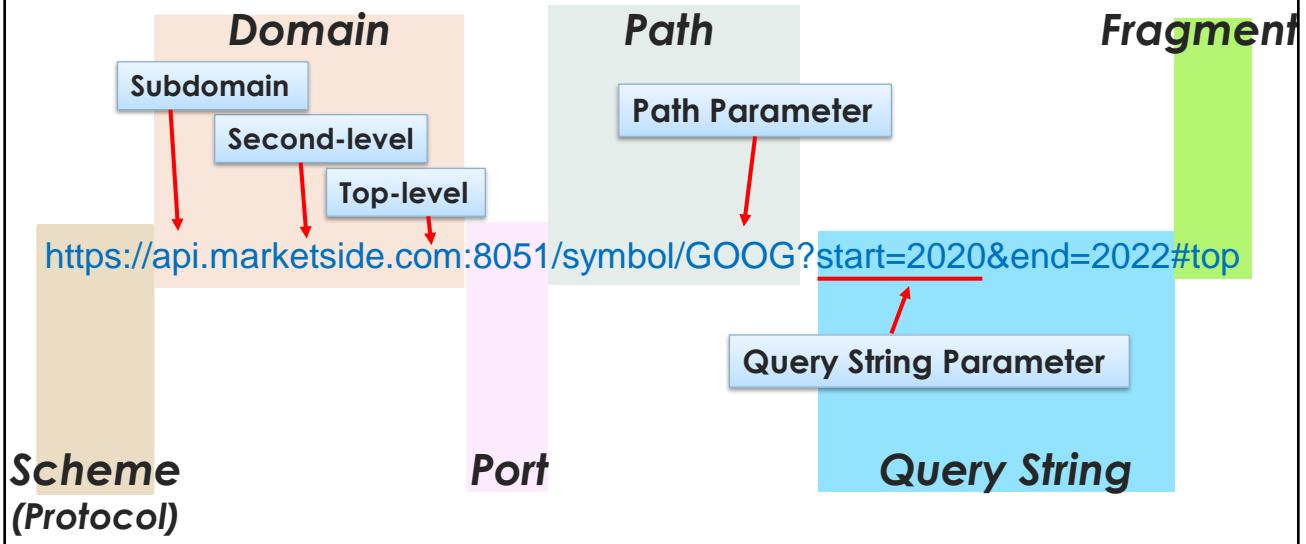
{
 customer_id: 121,
 name: Blackburn Industries,
 ...
}

- 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



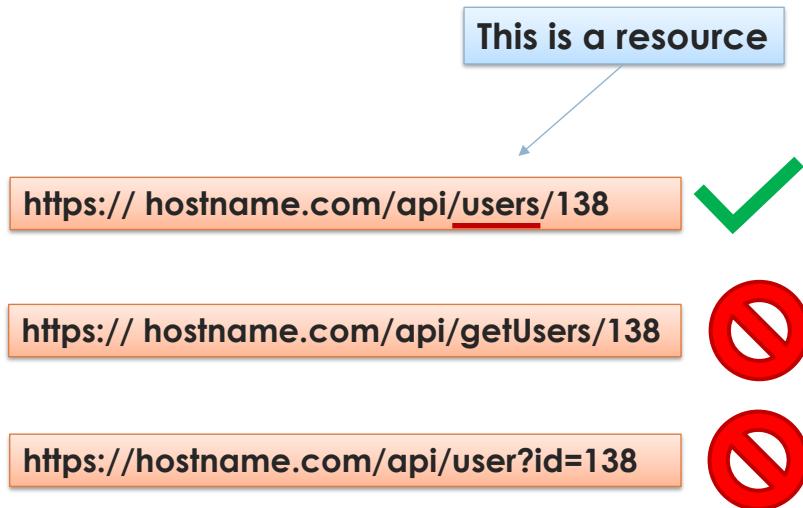
It will be important to distinguish between a path parameter and a query string parameter

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://hostname.com/api/users`). Resources are usually related to a domain or business objects from your application.

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

RESTful Design Principles (*continued*)

- Use HTTP methods to define operations

GET /api/invoices/**2952**

Reads a specific 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

GET /api/invoices

Retrieves all invoices

POST /api/invoices

Creates a new 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

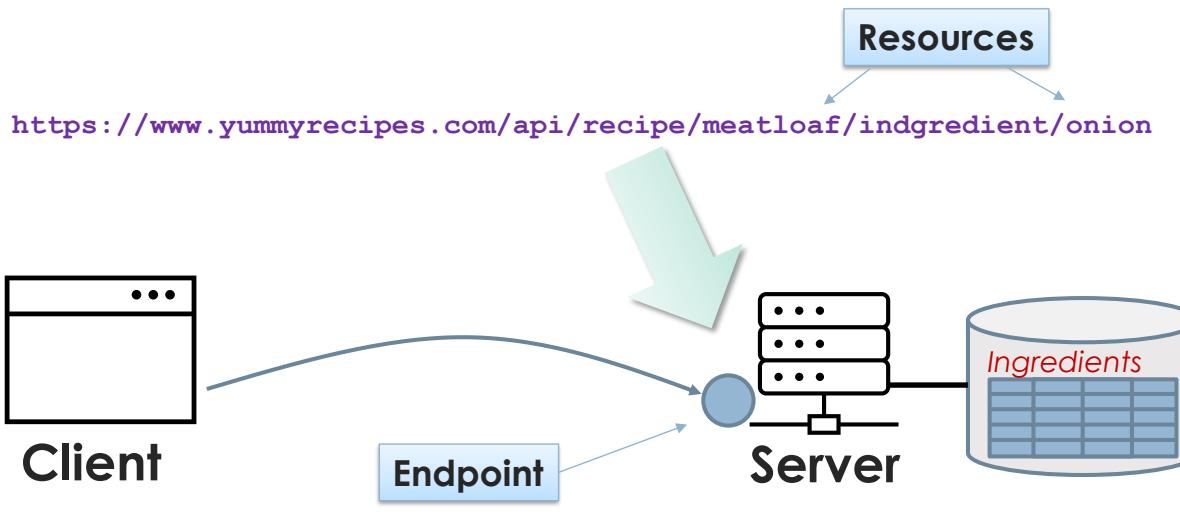
POST, PUT, PATCH, and DELETE can place parameters to be sent in the HTTP message body

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 a message sent using 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 a great language for developing APIs
 - 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

Flask

FastAPI

Django / Django REST

Falcon

Bottle

Client-side Tools

Requests

PycURL

Urllib (Standard library)

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/?start=1&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.")}')
```

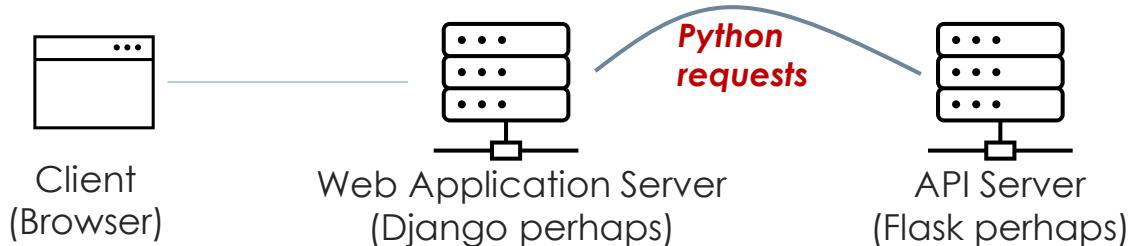
Helena, Helena, Helena

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.

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
 - *pip install* statements can vary:
pip3 install requests
python -m pip install requests

pip install requests

A basic *requests* module 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.

The params parameter of the `requests.get()` call enables passing query string parameters.

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.

The object returned from *requests.get()*, *post()*, *put()*, etc., is a response object (called r here).

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

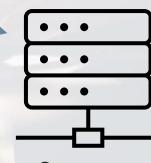
- Use the Python requests module to make a request and print the responses (either as raw text or as a dictionary) for the URLs used in Task 1-2
- *Work from the `task1_3_starter.py` file*

<https://api.coinlore.net/api/tickers/?start=1&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
requests**

Python-
based
Client



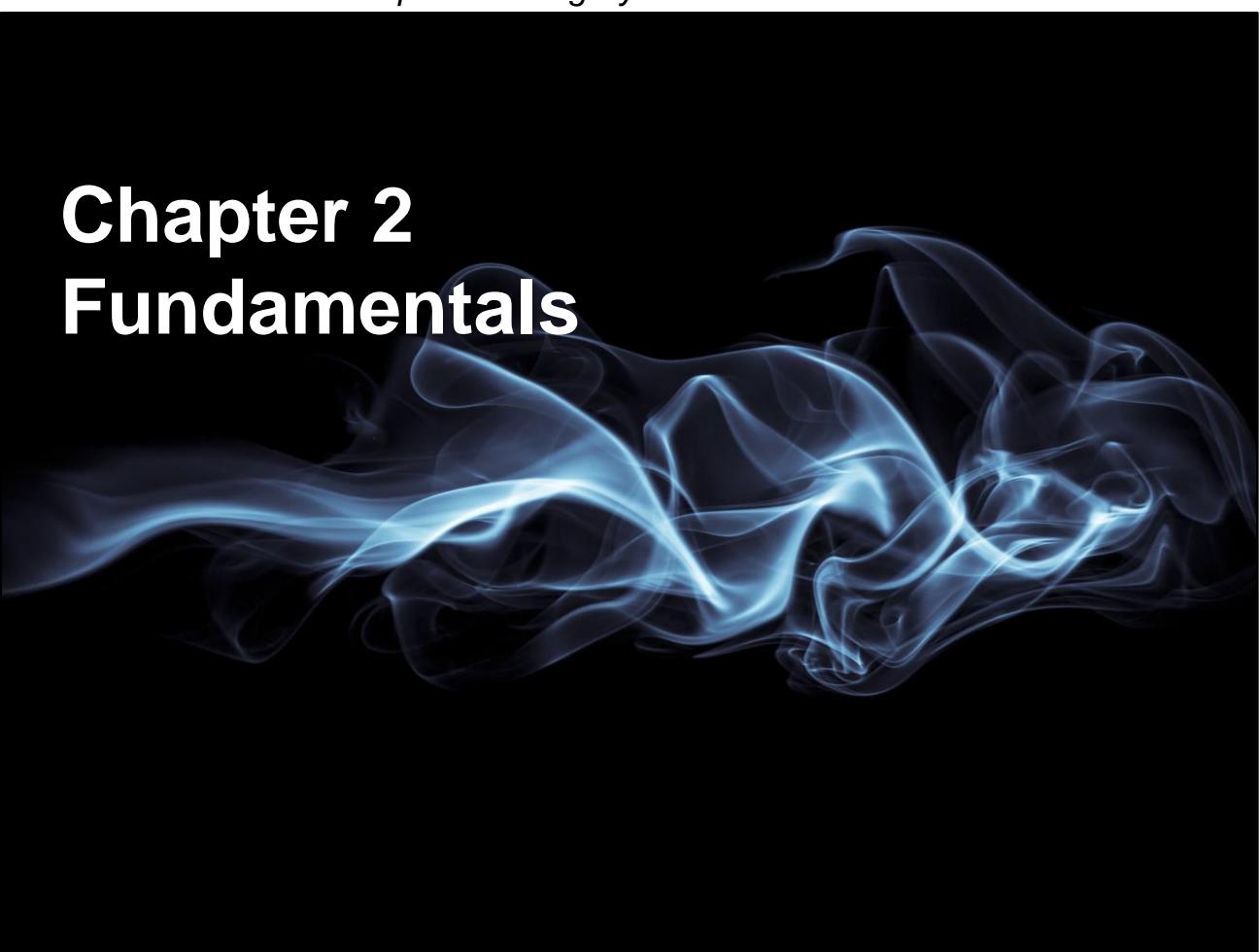
Server

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 often return JSON data to a client
 - 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, Django, and FastAPI

Chapter 2

Fundamentals



Chapter 2 Overview

API Fundamentals

Asyncio Overview

Introducing FastAPI

Creating an API

Incorporating a Database

Several Python API Server Choices

Advantages



FastAPI

Performance,
ASGI-based

Disadvantages

Newer, less community
support, few plugins



web development,
one drop at a time

Strong community
support, numerous
plugins

Older, updates tend to
cause compatibility
issues with plugins



All-in-one
solution

Bigger learning curve,
oldest of these three

- *Numerous others*

The brief summaries above understate some of the advantages and disadvantages of each server listed. While FastAPI is one of the newest entries to the crowded list of Python web servers, it has become popular quite quickly.

Users of FastAPI include Netflix, Microsoft, and many other multi-billion dollar companies. It was created and is owned by a single developer, Sebastian Ramirez, in 2018

Introducing FastAPI

- FastAPI is a newer addition to the list of popular Python tools for building RESTful applications
- Since it is a third-party tool, it must be installed first



`pip install fastapi[standard]`

<https://fastapi.tiangolo.com/>

- Fast API is based on *ASGI*...

For more on FastAPI, visit <https://fastapi.tiangolo.com/>.

The `fastapi[standard]` install includes FastAPI libraries plus Starlette, Pydantic, and uvicorn. Removing the `[standard]` declaration will leave out the optional dependencies mentioned.

What is ASGI?...

- In Python, an earlier standard, called WSGI, allowed web server vendors to standardize their solutions
 - **ASGI**, or Asynchronous Server Gateway Interface, is a modernized version of WSGI that emphasizes better concurrency
- <https://asgi.readthedocs.io/en/latest/>
- ASGI apps feature a single-thread used in a non-blocking mode so that it can perform more tasks than synchronous apps
 - It does this by deferring control of the thread back to an *event loop*--this is called **awaiting**

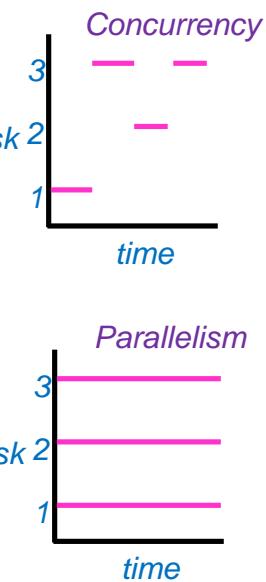
ASGI is a set of APIs in Python for handling concurrency via a single-threaded model. It attempts to use a single thread to a higher degree than a typical synchronous single-threaded application. For this to work properly, it means that single-thread must never block. When a task would normally want to block, control is returned to the event loop, via the *await* keyword, and the event loop can do other things during this time.

Concurrency vs Parallelism

- **Concurrency** involves running multiple tasks in a single CPU core, switching tasks as needed

- **Parallelism** - can schedule multiple tasks into multiple CPU cores to run simultaneously

- Python provides modules to help improve application performance
 - o **threading** - single-process, concurrency model, GIL
 - o **multiprocessing** - supports parallelism OS processes
 - o **asyncio** - supports single-threaded concurrency by switching tasks executed

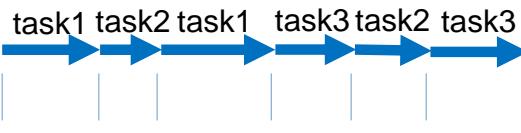


There are *high-level* asyncio APIs for general use and *low-level* asyncio API for those building libraries to gain better control over the event loop

Several options exist in Python for improving the performance of an application. The threading model suffers from having to share time with the CPU due to the presence of a global interpreter lock that only allows one thread at a time to actually execute. Multi-processing is a good way to achieve actual parallelism, but come as a cost of increased complexity. Asyncio is subject to the same constraints as the threading model, namely the global interpreter lock, but attempts to improve performance by utilizing that single thread to a higher degree.

Asynchronous Tasks

- **asyncio** executes multiple tasks in a *single thread* using time-slicing which allows tasks to share the CPU



- Tasks execute until they *yield* to another thread
 - Problematic if tasks are greedy
 - Works best with lots of I/O operations

```

import asyncio
import time

async def my_func(val: int):
    print(f'entering my_func {val}')
    await asyncio.sleep(1)
    print(f'exiting my_func {val}')

async def main():
    await asyncio.gather(my_func(1),
                         my_func(2))

start = time.time()
asyncio.run(main())
print(f'Elapsed: {time.time() - start}')
  
```

async def defines a "coroutine" or any function that will *yield* control

Yields control

Schedules coroutines into the event loop

Runs the event loop, passes the coroutine to it

```

entering my_func 1
entering my_func 2
exiting my_func 1
exiting my_func 2
Elapsed: 1.0113542079925537
  
```

student_files/ch02_fundamentals/01Coroutine.py

Asyncio assumes a single-thread is executing tasks but only allows tasks to run for a period. In some cases, this approach can be less complex than a multi-threaded implementation because we aren't as susceptible to thread synchronization problems. Remember, there is only one thread executing.

When compared to the single-threaded version, asynchronous solutions can be better when there is a lot of I/O blocking. Blocking occurs when the program is waiting for input, perhaps from outside the Python environment. In solutions where there is a lot of blocking, this approach can provide a performance improvement. Also, if there are a lot of tasks, it is possible that some tasks will likely involve some IO blocking.

Good reference: <https://github.com/timofurrer/awesome-asyncio>

Processing Items Asynchronously

```

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)
            print(f'{filepath.name}: {line.strip()}')

async def main(files: list[str]):
    files = (Path(f'..../resources/{file_obj}') for file_obj 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))

```

This line allows control to return to the event loop, thus files are processed 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,(202)421-9008
car_crashes.csv: 18.1,7.421,4.525,16.290000000000003,17.014,1053.48,133.93,AK
accounts.csv: 101,Sally Jones,6710.11,0.025,C
contacts.dat: John Brown,231 Oak Blvd. Black Hills SD 82101,(719)303-1219,

Files are read concurrently

[student_files/ch02_fundamentals/02_await.py](#)

The example takes three files and reads 10 lines from each file. This happens by putting three process_data() calls into the event loop. To read a file in the traditional way, however, would cause the function to never yield back to the event loop. Each file would be processed entirely and sequentially.

However, in this example, await asyncio.sleep(0) causes control to be returned to the event loop. The output then renders data from the files concurrently as a result.

Async Keywords to Help Yield Control

async with and **async for** are Python language modifications that support yielding control to the event loop

aiofiles is a third-party library that provides methods that help yield control to the event loop

```
import aiofiles
from asyncio import islice

async def process_data(filepath: Path):
    async with aiofiles.open(filepath) as f:
        async for line in islice(f, 10):
            print(f'{filepath.name}: {line.strip()}')

async def main(files: list[str]):
    files = (Path(f'..../resources/{file_obj}') for file_obj 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))
```

This is a third-party version of **islice()** that allows **asyncio** objects to be iterated over

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,(202)421-9008
...

student_files/ch02_fundamentals/03_aiofiles.py

Asyncio apps can get complicated, but the key to remember is they need to yield control to the event loop. New Python control structures are designed to assist in this effort, making **asyncio** solutions easier to write.

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 output above shows the contents of the *celebrity_100.csv* file. This file is found in the *data* directory within the *student_files* folder. We'll develop our initial API around this file.

Defining Resources

- As part of our design process, we must decide on what our resources will be
- Given the data on the prior slide, we'll use a *celebrity* object, and similarly a plural noun for our URI
- So, our likely URL will look like the following:

`http://localhost:8000/api/celebrities/celebrity_name`

Our initial design will be centered around our data file (celebrity_100.csv), therefore our initial resource will be a celebrity object. The URL that maps to this resource is shown above, where celebrity name will be an actual name, such as Tom Cruise.

Starting the Server

- Our first version instantiates the FastAPI server object and the uvicorn tool starts it

Defines the main FastAPI application object

Starts the server

```
import uvicorn  
from fastapi import FastAPI  
  
app = FastAPI()  
  
uvicorn.run(app, host='localhost', port=8000)
```

Example startup within PyCharm:



Note: To run in debug mode, right-click the source code and select Debug...

`student_files/ch02_fundamentals/04_simple_api.py`

There's not much to test out yet. When the server is started within PyCharm, the output above will be shown.

Loading Some Data to Work With

- The app acquires data from a file at startup

```
from pathlib import Path

import uvicorn
from fastapi import FastAPI

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

app = FastAPI()

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

```
[['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/05_acquiring_data.py

In this updated version of our app, no new FastAPI code was added. Instead, we acquire 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

- Critical for working with FastAPI is to define which function should be called when a URI request arrives

```
from pathlib import Path

import uvicorn
from fastapi import FastAPI

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

app = FastAPI()

@app.get('/api/celebrities/{name}')
def do_stuff(name):
    return f'Hi there {name}!'

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

This statement responsible for registering the function within FastAPI. It tells FastAPI to call the function if a GET request matching this path is encountered

student_files/ch02_fundamentals/06_creating_a_mapping.py

The `app.get()` decorator registers the function within FastAPI and establishes when the function should be called. In addition, it can support other important parameters (some discussed later) such as `response_model` and `status_code`.

Defining a Mapping (continued)

- The `app.get()` path operation decorator gives FastAPI control over your functions

```
from pathlib import Path

import uvicorn
from fastapi import FastAPI

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

print('Celebrity data read.')

app = FastAPI()

@app.get('/api/celebrities/{name}')
def do_stuff(name):
    return f'Hi there {name}!'

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

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

But what is this {name} item?

student_files/ch02_fundamentals/06_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, FastAPI is replacing our `do_stuff()` function with another function of their own.

FastAPI Path Operation Decorators

- *Path operation decorators* are used above a path operation function to instruct FastAPI on how to call it

```
@app.get('/api/state/{st_name}/city/{city_name}')  
def get_population(st_name, city_name):
```

```
...
```

Items inside the "curlies" indicate path parameters

Variables should be provided in your function (operation) that match these path parameters

```
@app.get('/api/email/{id}')  
def read_email (id: int):
```

```
...
```

FastAPI will "inject" these values into your function for you and call your function

FastAPI can convert (type cast) the parameters based on the type hint specified in the function

The curly brackets, { }, indicate path parameters. They 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. Query string parameters are discussed later.

FastAPI will inject the path parameters into your function, so it is important to specify variables in the function that match the variables defined in the URL. Type hints may also be specified in the function and FastAPI will attempt to convert them to that type (this functionality comes from a helper tool called Pydantic, discussed further later).

A Raw Output Test

- To test out our partial implementation, first run the Python script
 - If using PyCharm, right-click anywhere in the source file and select
`Run '06_creating_a_mapping.py'`
- Next, in a browser type:
`http://localhost:8000/api/celebrities/tom%20cruise`
- You should get a response
Hi there tom cruise!

**Note: what happens if you type the URL, but leave off the celebrity name?
(e.g., `http://localhost:8000/api/celebrities/`)**

Note: FastAPI can also be run from the command-line by removing the `unicorn.run()` statement and typing the following in the terminal: `fastapi dev 06_create_a_mapping.py`

Other Output Formats

- The browser can only execute HTTP GET requests from its address bar
- If other HTTP methods are created, they will have to be tested differently
- Test out the following URLs in the browser:

`http://localhost:8000/openapi.json`

`http://localhost:8000/docs`

`http://localhost:8000/redoc`

These are discussed further later

More will be said about these URLs and output formats a little later.

Creating a JSON Response

```
from pathlib import Path

import uvicorn
from fastapi import FastAPI

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

print('Celebrity data read.')

app = FastAPI()

@app.get('/api/celebrities/{name}')
def do_stuff(name):
    other = ['stuff1', 'stuff2']
    return {'greeting': f'Hi there {name}', 'more': other, 'anything': {'level': 2}}

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



```
{
  "greeting": "Hi there tom cruise",
  "more": [
    "stuff1",
    "stuff2"
  ],
  "anything": {
    "level": 2
  }
}
```

student_files/ch02_fundamentals/07_working_with_json.py

FastAPI naturally returns data as JSON objects. So, if you merely return a string, it will try to convert the string to a JSON string which requires certain characters to be escaped (such as double quotes). If you wanted to return something else, you can change the `response_model` or `response_class`. Below shows how to return data as plain text:

```
from fastapi.responses import PlainTextResponse
```

```
@app.get('path', response_class=PlainTextResponse)
```

GET Single Celebrity

```

from pathlib import Path

import uvicorn
from fastapi import FastAPI

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

app = FastAPI()

@app.get('/api/celebrities/{name}')
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

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

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

```

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

student_files/ch02_fundamentals/08_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.

FastAPI Error Handling

```
from pathlib import Path

import uvicorn
from fastapi import FastAPI, HTTPException

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

app = FastAPI()

@app.get('/api/celebrities/{name}')
def get_one_celebrity(name):
    results = [row for row in data if name.casefold() in row[0].casefold()]

    if not results:
        raise HTTPException(status_code=404, detail='Celebrity not found')

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

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

Our previous example responds with a 200 HTTP status even if no celebrity is found (e.g., try XYZ for the name)

Here we see FastAPI allows for raising exceptions that respond with an HTTP status and message of your choosing

student_files/ch02_fundamentals/09_get_single_celebrity_error_handling.py

This version improves on the previous one by adding error handling that supports changing the HTTP status code. A 404 status code is often selected when the resource sought is not found or identified.

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 FastAPI 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/10_consuming_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('Request 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/11_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 FastAPI server from task 2-1 is running before testing your answer
 - Access invoice `536365` and display its results

Passing Data in RESTful Methods

- Different HTTP methods will follow slightly different ways of passing data:

Request Type

GET /api/invoices/**2952**

Reads a specific invoice

POST /api/invoices

Creates a new invoice

PUT /api/invoices/**2952**

Updates a specific invoice

DELETE /api/invoices/**2952**

Deletes a specific invoice

Data Passed

Invoice ID (or values) passed in path only

Invoice data passed without an ID in the body of the HTTP method

Invoice ID passed in path, invoice data values passed in body

Invoice ID passed in path only, body can be empty

Passing data for an HTTP PATCH is similar to an HTTP PUT. Also, an HTTP GET that does not include a specific ID could be used to retrieve *many* records.

Python Classes

```

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')
header = f.readline().strip().lower().split(',')
header[1] = re.sub(r'\([^\)]*\)', "", header[1]).strip()      # regex removes "(millions)"
                                                               # ['name', 'pay', 'year', 'category']
data = f.readline().strip().split(',')                      # ['Oprah Winfrey', '225.0', '2005', 'Personalities']

data_dict = {h: d for h, d in zip(header, data)}          # {'name': 'Oprah Winfrey', 'pay': '225.0',
                                                               # 'year': '2005', 'category': 'Personalities'}
celeb = Celebrity(**data_dict)
print(celeb)                                              2005 Oprah Winfrey (Personalities) $225.0 million

```

student_files/ch02_fundamentals/12_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).

The `**` notation is used to expand (unpack) the dictionary into keyword arguments.

Python Annotations (Type Hints)

- Annotations provide developers and tools a means for

```
from pydantic import validate_call

@validate_call
def read_celebs(filepath: str | Path,
                 size: int = 10,
                 encoding: str = 'utf-8') -> list[Celebrity]:
    results = []
    with open(Path(filepath), encoding=encoding) as f:
        f.readline()
        for line in islice(f, size):
            values = line.strip().split(',')
            results.append(Celebrity(values[0], float(values[1]),
                                      int(values[2]), values[3]))
```

return results

determining what types to pass into a function

```
print(read_celebs(datafile, 7))
print(read_celebs(size='7', filepath=datafile))
print(read_celebs(datafile, 'hello'))
```

Okay
Okay, Pydantic will coerce string to int safely
Not okay, Pydantic can't convert 'hello' to int

student_files/ch02_fundamentals/13_python_type_hints.py

Shown here is the use of Python annotations. Annotations are a vital part of FastAPI's success.

Also shown is a framework called Pydantic. Pydantic is a popular third-party data validation framework. It relies on type hints within Python to work properly. In the example, a decorator gains control of the user function (`read_definitions()`) and checks function call parameters *before* the function actually gets called. If the parameters are invalid types, a `ValidationError` is raised.

Pydantic can convert string types to int if desired. This happens in the second call to `read_definitions()`. If it can't make that conversion, a `ValidationError` will occur.

Pydantic

- Pydantic provides runtime type validation (and more)

```
import sys
from pydantic import BaseModel, ValidationError
```

```
class Celebrity(BaseModel):
    name: str
    pay: float
    year: int
    category: str
```

Pydantic models should inherit from **BaseModel**

Like a Dataclass, declare attributes within the class, but not within an `__init__()`

Use keyword args when instantiating Pydantic model objects

```
c1 = Celebrity(name='Oprah', pay=225.0, year=2005, category='Personalities')
c2 = Celebrity(name='Oprah', pay=225.0, year='2005', category='Personalities')
try:
    c3 = Celebrity(name='Oprah', pay=225.0, year='hello', category='Personalities')
except ValidationError as err:
    print(err, file=sys.stderr)
```

Fails at runtime

student_files/ch02_fundamentals/14_using_pydantic.py

In the example, three celebrity objects are instantiated. Being Pydantic models, they should only use keyword arguments for creation. The first two will work. The second works because Pydantic can coerce the string provided for the year. The third fails at runtime during instantiation and raises a `ValidationError`.

Pydantic, FastAPI, and POST Form Data

```
from typing import Annotated
from fastapi import FastAPI, Form
app = FastAPI()

@app.post('/api/celebrities')
def create_celebrity(name: Annotated[str, Form()],
                      pay: Annotated[float, Form()],
                      year: Annotated[int, Form()],
                      category: Annotated[str, Form()]):
    return {'action': 'POST response',
            'name': name, 'pay': pay, 'year': year, 'category': category}
```

FastAPI

The server is told to call `create_celebrity()` on a POST request

Client

```
results = requests.post(f'{base_url}{path}'.strip('/'),
                        data={'name': celeb_name,
                               'category': 'Actors',
                               'pay': 3.0,
                               'year': 2024})
print(results.text)
print(results.request.body)
```

The client send form encoded data using the `data=` attribute

{"action": "POST response", "name": "Kevin",
"pay": 3.0, "year": 2024, "category": "Actors"}

name=Kevin&category=Actors&pay=3.0&year=2024

*student_files/ch02_fundamentals/15_post_as_form_data.py and
16_testing_post_as_form_send.py*

There are actually two files depicted here. The upper (first) is the FastAPI stub for the POST method. This method is told to expect data coming in as form encoded data values

Pydantic, FastAPI, and POST JSON Data

```
from typing import Annotated
```

FastAPI

```
from fastapi import Body, FastAPI
```

The server is told to call `create_celebrity()` on a POST request

```
app = FastAPI()
```

```
@app.post('/api/celebrities')
def create_celebrity(name: Annotated[str, Body()],
                     pay: Annotated[float, Body()],
                     year: Annotated[int, Body()],
                     category: Annotated[str, Body()]):
    return {'action': 'POST response',
            'name': name, 'pay': pay, 'year': year, 'category': category}
```

Data is expected to arrive as form encoded data values

Client

```
results = requests.post(f'{base_url}{path}'.strip('/'),
                        json={'name': celeb_name,
                               'category': 'Actors',
                               'pay': 3.0,
                               'year': 2024})
```

The client send form encoded data using the `data=` attribute

```
print(results.text)
print(results.request.body)
```

{"action": "POST response", "name": "Kevin",
"pay": 3.0, "year": 2024, "category": "Actors"}

b'{"name": "Kevin", "category": "Actors", "pay": 3.0, "year": 2024}'

*student_files/ch02_fundamentals/17_post_as_json_data.py and
18_testing_post_as_json_send.py*

This version is nearly identical to the previous except the HTTP body of the request will send the data in a JSON format.

Pydantic Models in POST

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

FastAPI

```
class Celebrity(BaseModel):
    name: str
    pay: float
    year: int
    category: str
```

By setting any of these = None as a default value, they become optional to provide

```
app = FastAPI()

@app.post('/api/celebrities')
def create_celebrity(celebrity: Celebrity):
    return {'action': 'POST response', **celebrity.model_dump()}
```

JSON data in the HTTP body is gathered and placed into the Celebrity object

```
results = requests.post(f'{base_url}{path}'.strip('/'),
                       json={'name': celeb_name, 'category': 'Actors',
                             'pay': 3.0, 'year': 2024})
print(results.text)
print(results.request.body)
```

Client

{"action": "POST response", "name": "Kevin",
"pay": 3.0, "year": 2024, "category": "Actors"}

b'{"name": "Kevin", "category": "Actors", "pay": 3.0, "year": 2024}'

*student_files/ch02_fundamentals/19_post_using_pydantic_models.py and
20_testing_post_using_pydantic_models.py*

While the client still sends data as JSON in the HTTP body, the server is told to expect a Celebrity (Pydantic model) object. FastAPI extracts the data from the HTTP body and creates a Celebrity object. It does Pydantic runtime type validation as well as injection of this object into the create_celebrity() method.

Providing the Other HTTP Methods

```
@app.get('/api/celebrities')
def get_all_celebrities():
    return {'action': 'GET (all) response'}
```



```
@app.put('/api/celebrities/{name}')
def update_celebrity():
    return {'action': 'PUT response'}
```



```
@app.delete('/api/celebrities/{name}')
def delete_celebrity():
    return {'action': 'DELETE response'}
```

FastAPI

These are the other HTTP methods
(though they are not finished yet)

```
results = requests.put(f'{base_url}{path}{celeb_name}')
print(results.text) {"action": "PUT response"}
```

```
results = requests.delete(f'{base_url}{path}{celeb_name}')
print(results.text) {"action": "DELETE response"}
```

```
results = requests.get(f'{base_url}{path}'.strip('/'))
print(results.text) {"action": "GET (all) response"}
```

The client issuing other HTTP
requests (though incomplete
also) to the server

Client

*student_files/ch02_fundamentals/19_post_using_pydantic_models.py and
20_testing_post_using_pydantic_models.py*

Your Turn! - Task 2-3

- Incorporate the POST, PUT, DELETE, and GET (all) into our previous invoice application (Task 2-1)
 - Work from the newly provided `task2_3_starter.py` file
- Create an *Invoice* Pydantic Model which defines a `stock_code` (str), `quantity` (int), `description` (str), `invoice_date` (str), `unit_price` (float), `customer_id` (str), and `country` (str)
- Write the function (operation) signatures and decorators for each of the above mentioned HTTP methods
- Place temporary return values in each of the functions
- Use the provided test client to test your solution (`task2_3_client.py`)

Optional Topic: 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

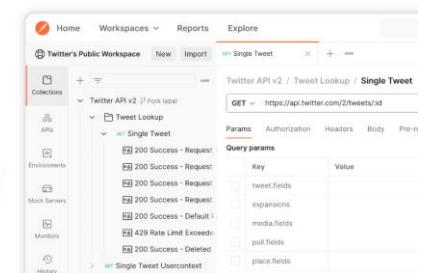
- As a desktop app installed from

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

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

Download Postman

Download the app to quickly get started using the Postman API Platform. Or, if you prefer a browser experience, you can try the new web version of Postman.



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

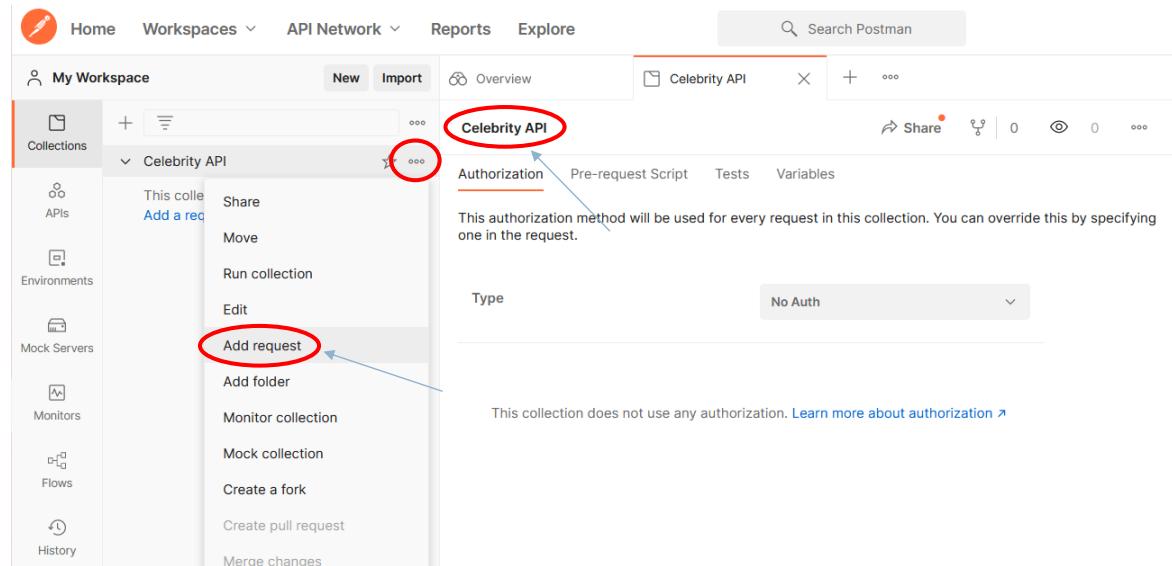
The screenshot shows the Postman web application interface. At the top, there is a navigation bar with links for Home, Workspaces, API Network, Reports, Explore, and a search bar labeled 'Search Postman'. To the right of the search bar are several icons: a green gear, a blue 'Invite' button, a gear, a bell, and a refresh symbol. Below the navigation bar is a header titled 'My Workspace' with sub-links for Overview, New, Import, and a workspace settings icon.

The main content area is titled 'My Workspace' and contains the following sections:

- Add a brief summary about this workspace:** A text input field for describing the workspace.
- This is your personal, private workspace to play around in. Only you can see the collections and APIs you create here - unless you share them with your team.** A descriptive message about the workspace's purpose.
- Activity:** A section showing recent activity with filter options for 'Filter by', 'Elements', and 'People'.
- In this workspace:** A sidebar listing categories with their counts:
 - Requests (0)
 - Collections (0)
 - APIs (0)
 - Environments (0)
 - Mock Servers (0)
 - Monitors (0)
- Create a collection for your requests:** A section with a description and a 'Create collection' button.
- Media Authentication API:** A collapsed collection containing several requests:
 - POST Authentication
 - POST Access Token
 - GET User's Page
 - GET Instagram Account
 - GET New Request
- Sports team fixtures:** A collapsed collection under 'Sports Arena'.

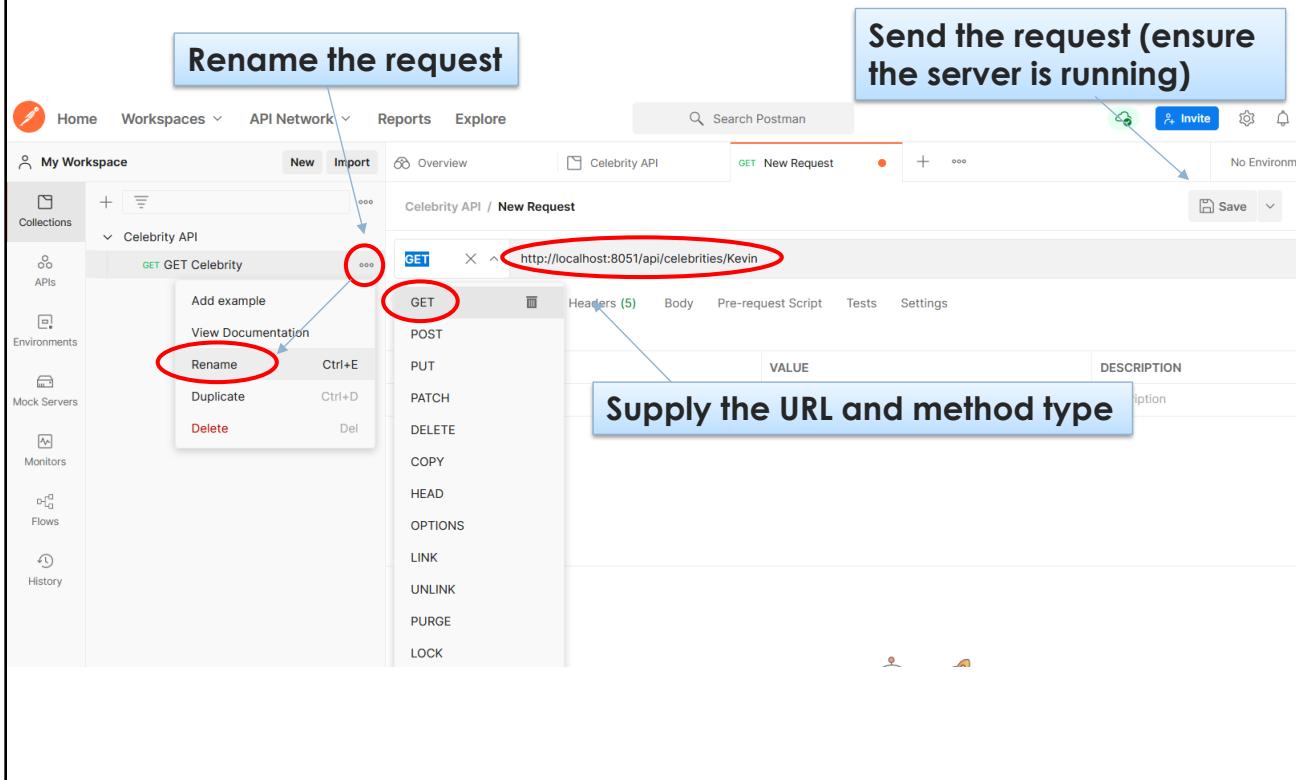
On the screen shown above, the browser-based version loaded the **web.postman.co** url and signed in. Next, a collection was created. Collections are groupings of URLs (typically related to an API set).

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, highlighted with a red circle. A table titled 'Query Params' shows one entry: 'foo' with 'bar' as its value. Other tabs include Authorization, Headers (5), Body, Pre-request Script, Tests, and Settings.

Optionally set headers

View results

Send the request

View results

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 items: "GET GET Invoices", "POST POST Invoices", "GET GET Invoice", and "PUT PUT Invoice". The "PUT PUT Invoice" item is currently selected. Below it, a "Query Params" table has one row with "Key" and "Value" columns. The "Body" tab is selected, showing a JSON response:

```

{
  "results": [
    {
      "id": "85123A",
      "name": "WHITE HANGING HEART T-LIGHT HOLDER",
      "category": "6",
      "price": "2.55",
      "rating": "5.0",
      "country": "United Kingdom"
    },
    {
      "id": "71053",
      "name": "WHITE METAL LANTERN",
      "category": "6",
      "price": "2.75",
      "rating": "4.8",
      "country": "United Kingdom"
    },
    {
      "id": "84406B",
      "name": "CREAM CUPID HEARTS COAT HANGER",
      "category": "8",
      "price": "3.99",
      "rating": "4.5",
      "country": "United Kingdom"
    },
    {
      "id": "840296",
      "name": "KNITTED UNION FLAG HOT WATER BOTTLE",
      "category": "6",
      "price": "3.99",
      "rating": "4.2",
      "country": "United Kingdom"
    },
    {
      "id": "84029E",
      "name": "RED WOOLLY HOTTIE WHITE HEART",
      "category": "6",
      "price": "3.99",
      "rating": "4.0",
      "country": "United Kingdom"
    },
    {
      "id": "22752",
      "name": "SET 7 BABUSHKA NESTING BOXES",
      "category": "2",
      "price": "7.65",
      "rating": "4.5",
      "country": "United Kingdom"
    },
    {
      "id": "21730",
      "name": "GLASS STAR FROSTED T-LIGHT HOLDER",
      "category": "6",
      "price": "4.25",
      "rating": "4.8",
      "country": "United Kingdom"
    }
  ]
}
  
```

The status bar at the bottom indicates: Status: 200 OK Time: 46 ms Size: 885 B Save Response.

Bringing In the Database

- A commonly used Python tool for interacting with a database is called **SQLAlchemy**

<https://docs.sqlalchemy.org/en/20/index.html>

- SQLAlchemy performs *automatic Python object-to-relational database mapping* (ORM)
- SQLAlchemy is a third-party tool and must be installed

`pip install sqlalchemy`

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.

Configuring SQLAlchemy

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

- Step 1.**

Configure the tool via `create_engine()` which defines how to connect to the database

```
from pydantic import BaseModel
from sqlalchemy import create_engine, Column
from sqlalchemy.orm import declarative_base, sessionmaker
from sqlalchemy.types import Float, Integer, String

db_url = Path(__file__).parents[1] / 'data/course_data.db'
app = FastAPI() This is the location of our database

engine = create_engine('sqlite:///{} + str(db_url), echo=True)
Session = sessionmaker(bind=engine)
session = Session()
```

student_files/ch02_fundamentals/21_fastapi_sqlalchemy.py

To configure SQLAlchemy, we call `create_engine()` passing a preconfigured string (URL) to identify how to connect to the database. SQLAlchemy doesn't actually connect until it needs to. For other databases, other properties may be required, such as properties for a username and password to authenticate with the database.

A session is also defined which will be used to perform the SQL operations.

For more on SQLAlchemy connection strings, visit the following URL:
<https://docs.sqlalchemy.org/en/20/core/engines.html>

Define SQLAlchemy Models

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

```
app = FastAPI()

Base = declarative_base()

class CelebrityModel(Base):
    __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: str, pay: float, year: str, category: str):
        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 Base

Define the fields and types to map to the table

The `__init__` and `__str__` are optional but can be helpful

`student_files/ch02_fundamentals/21_fastapi_sqlalchemy.py`

A model class should inherit from `Base` (which is a SQLAlchemy-derived 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 FastAPI & SQLAlchemy

- Step 3.

We'll modify our POST path parameter operation to support SQLAlchemy

```
@app.post('/api/celebrities')
async def create_celebrity(new_celeb_request: Celebrity):
    try:
        new_celeb_db =
            CelebrityModel(**new_celeb_request.model_dump())
        session.add(new_celeb_db)
        session.commit()
        results = new_celeb_db.to_dict()
    except Exception as err:
        raise HTTPException(status_code=404,
                            detail=f'Error creating celebrity: {err.args[0]}')
    return results
```

Instantiate a model, perform an INSERT

Commonly, the inserted object is returned

Test the POST operation either using Postman or by running
22_testing_fastapi_sqlalchemy.py

Our POST method of the Celebrities class has been modified. It now extracts the HTTP request (body) parameters placing them into a new_celeb_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.

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.

Your Turn! - Task 2-5:

Implement GET (singular) and POST

- Add [SQLAlchemy](#) to the current Invoice API
- Complete the API's POST method to use it
- Work from the provided `task2_5_starter.py` file
- Test it by running the provided `task2_5_client.py`

Implementing GET (All) SQLAlchemy

```
@app.get('/api/celebrities/{name}')
def get_all_celebrities(name: str):
    celebs = session.query(CelebrityModel).filter(CelebrityModel.name.contains(name)).all()
    if not celebs:
        raise HTTPException(status_code=404, detail=f'No celebrities found.')
    return celebs
```

Performs equivalent to a SELECT * FROM celebrity WHERE

Return all records that contain the name parameter within the name column

student_files/ch02_fundamentals/21_fastapi_sqlalchemy.py

Now that the pieces are in place, the remaining methods should come along easily. The get() (all) method is shown above. This returns matches to a celebrity name.

Implementing PUT with SQLAlchemy

```

@app.put('/api/celebrities/id/{celeb_id}')
def update_celebrity(celeb_id: int, updated_celeb: Celebrity):
    try:
        celeb = session.get(CelebrityModel, celeb_id) ← Using the object id, lookup the
                                                       object in the database, then use
                                                       the object to update it

        celeb.name = updated_celeb.name if updated_celeb.name else celeb.name
        celeb.year = updated_celeb.year if updated_celeb.year else celeb.year
        celeb.category = updated_celeb.category if updated_celeb.category else celeb.category
        celeb.pay = updated_celeb.pay if updated_celeb.pay else celeb.pay

        session.commit() ← Make any changes (e.g.,
                           celeb.year = year) to our model
                           based on the updated_celeb

        results = celeb.to_dict()
    except Exception as err:
        raise HTTPException(status_code=404, detail=f'Error updating celebrity: {err.args[0]}')

    return results

```

student_files/ch02_fundamentals/21_fastapi_sqlalchemy.py

The put() is implemented now.

Implementing DELETE with SQLAlchemy

```
@app.delete('/api/celebrities/id/{celeb_id}')
def delete_celebrity(celeb_id: int):
    try:
        celeb = session.get(CelebrityModel, celeb_id)
        session.delete(celeb)
        session.commit()

        results = celeb.to_dict()
    except Exception as err:
        raise HTTPException(status_code=404, detail=f'Error removing celebrity: {err.args[0]}')

    return results
```

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

student_files/ch02_fundamentals/21_fastapi_sqlalchemy.py

The `delete_celebrity()` is implemented now and shown above.

Your Turn! - Task 2-6

- Complete the Invoice API
- Add the remaining API methods: `get()` all, `put()`, and `delete()`
- Work from your provided `task2_6_starter.py` file
- Test it by running the provided `task2_6_client.py` file or by running it within
`http://localhost:8000/docs`

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	Bad Request (Usually no URL mapping)
401	Unauthorized (Usually when client is not authenticated)
403	Forbidden (Usually when client is not authorized but is authenticated)
404	Not found (Good when resource id isn't found)
405	Method not allowed (e.g., A POST occurred on a GET only method)
429	Too many requests (User sent too many requests in a given period of time)
500	Internal Server Error (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	Ok (The majority of response status codes are 200)
201	Created (The server created a new resource)
204	No Content (The request processed successfully, but the server didn't have anything to respond with)
300	Multiple Choices (The request has multiple possible responses)
304	Not modified (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.

Chapter 2 Summary

- Python provides tools to facilitate building all aspects of APIs
- **FastAPI** is only one of many options for serving content
- The FastAPI environment provides a modern approach that uses ASGI to perform at competitive speeds to what other languages have to offer
- FastAPI can work with database by integrating with SQLAlchemy
 - Using SQLAlchemy within FastAPI is optional, however

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

    def get(self):
        page = request.args.get('page') or '1'
        query = db.select(CelebrityModel).order_by(CelebrityModel.pay.desc())
        paged_celebs = db.paginate(query, page=int(page),
                                    per_page=self.page_size, error_out=False).items
        return {'results': celebrities_schema.dump(paged_celebs.items)}
    
```

Pages are 10 records

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

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

```

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)
    
```

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

`pip install flask-limiter`

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

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from pathlib import Path

app = Flask(__name__)
limiter = Limiter(get_remote_address, app=app)

@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

To 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)

```
swagger: "2.0"
```

```
info:
```

```
  title: Generated Flask Server
```

```
paths:
```

```
  '/celebrity/{name}':
```

```
    get:
```

```
      operationId: api.get_celebrity
```

```
      parameters:
```

- name: name

- description: Celebrity name

- in: path

- type: string

- required: true

```
      responses:
```

```
        200:
```

```
          description: Response to a celebrity query
```

```
          schema:
```

- type: object

- properties:

- message:**

- type:** string

- status:**

- type:** integer

swagger.yaml

Our URL mapping: <http://localhost:8051/celebrity/Kevin>

get_celebrity() function within `api.py`

`get_celebrity(name)` takes a **name** parameter

It will be defined in the path

It returns an object with *message* & *status* props

See source file for more

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 otherwise 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, a section titled 'Generated Flask Server' contains the text 'Example Generating Flask Server From Swagger YAML'. To the right, a callout box says 'With the server running, navigate to: http://localhost:8051/ui'. The main content area is titled 'default' and lists several API operations:

- GET /celebrity**
- POST /celebrity**
- DELETE /celebrity/{name}**
- GET /celebrity/{name}**
- PUT /celebrity/{name}**

Below the operations, a note states '[BASE URL: , API VERSION: 1.0.0]'. A callout box highlights the five operations 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://hostname.com/customers/17850",
      "action": "GET",
      "types": ["text/xml", "application/json"]
    },
    {
      "rel": "customer",
      "href": "https://hostname.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

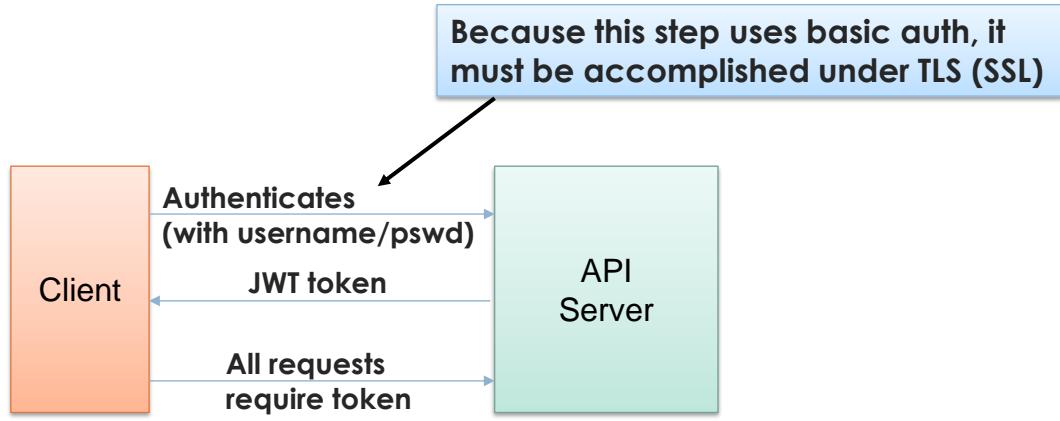
Basic authentication is secure only if you always make sure to use TLS (SSL).

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

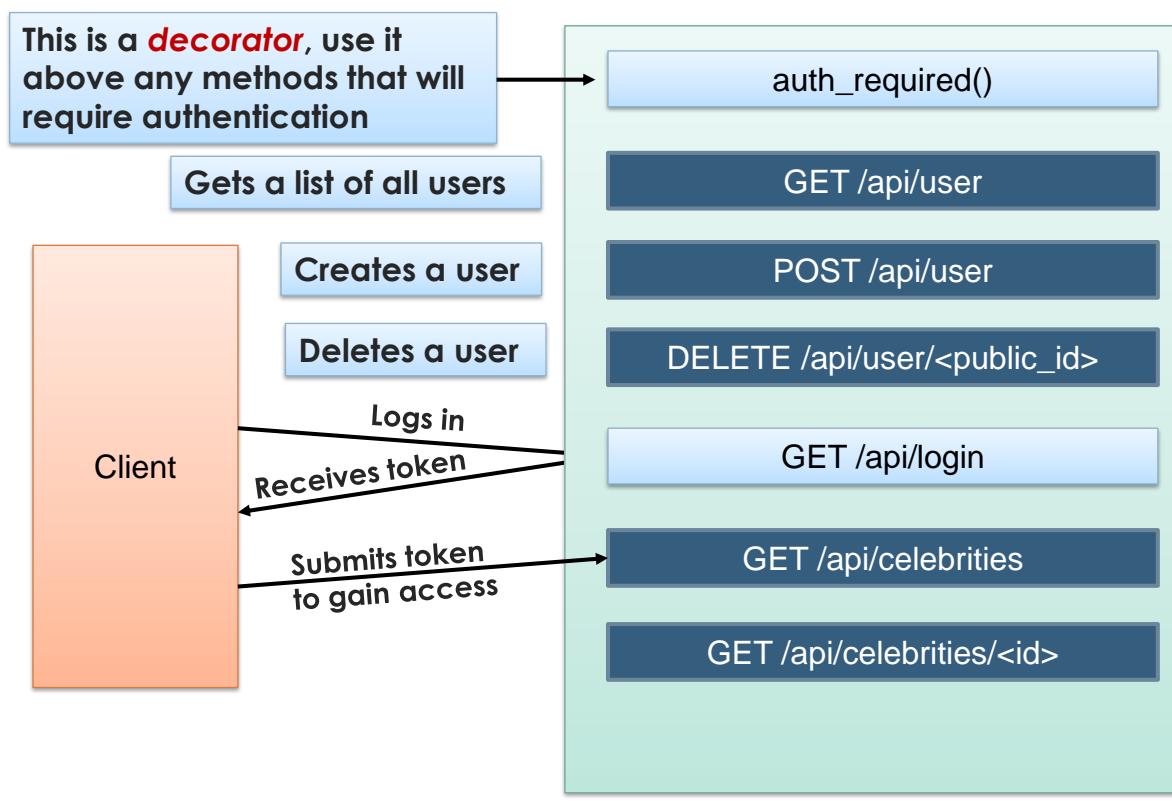
- JSON Web Tokens (JWTs) can be a useful means for securing APIs



- 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()
```

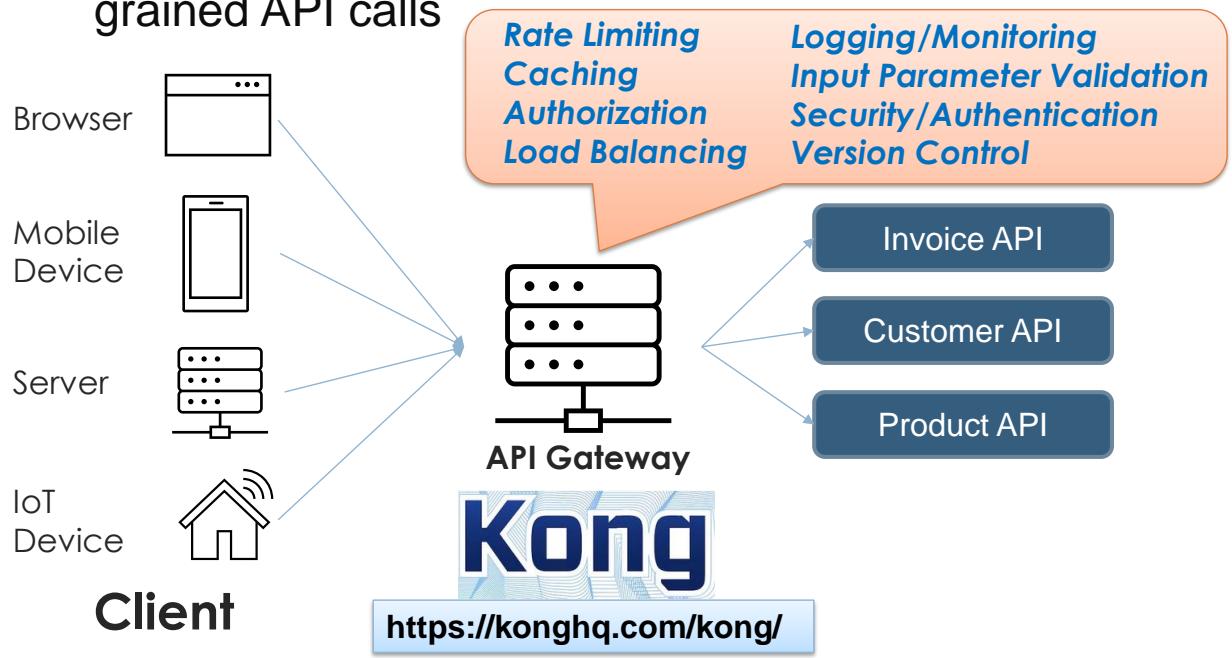
The test client (not shown) performs the following steps:

- 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

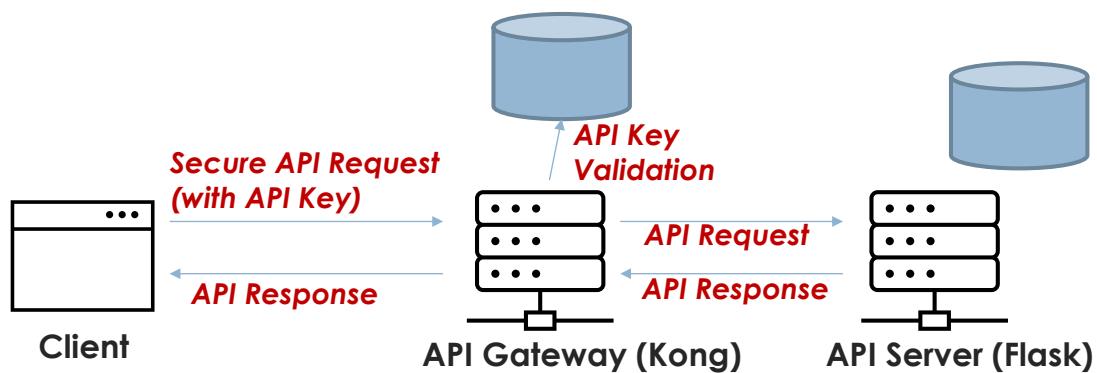


API gateways assist with providing a single interface in front of a microservices architecture thus simplifying client knowledge of the API network.

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

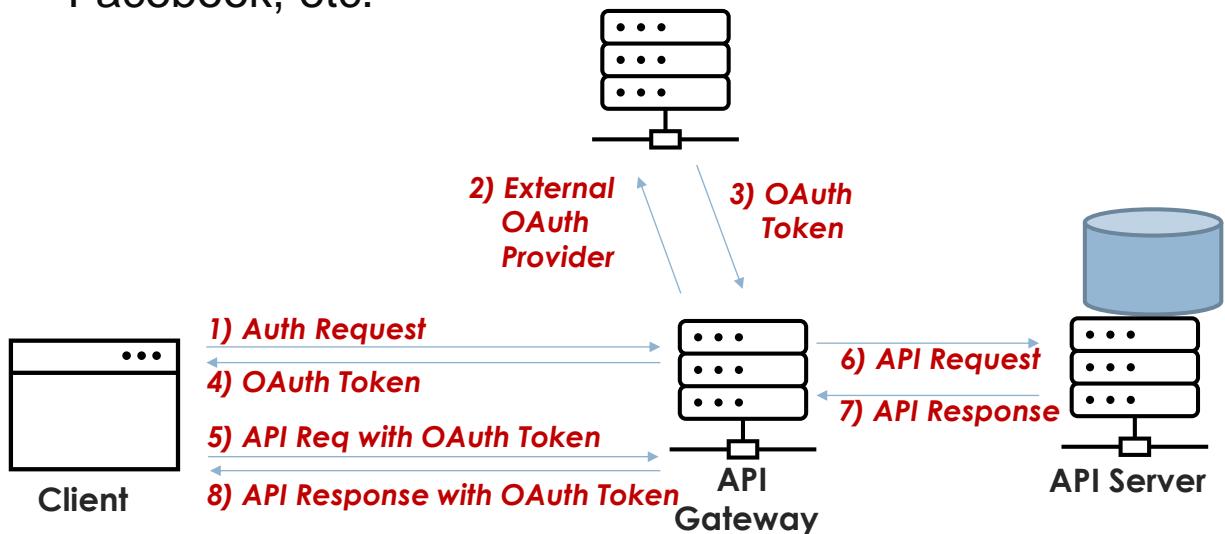
- API keys provide a means to link API access to a specific client
 - Usually, a client registers for an API key
 - The key is submitted for every GET request which identifies the client
 - Resource access and rate limits can then be applied
 - Common within public (non-private) APIs
 - API keys are 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

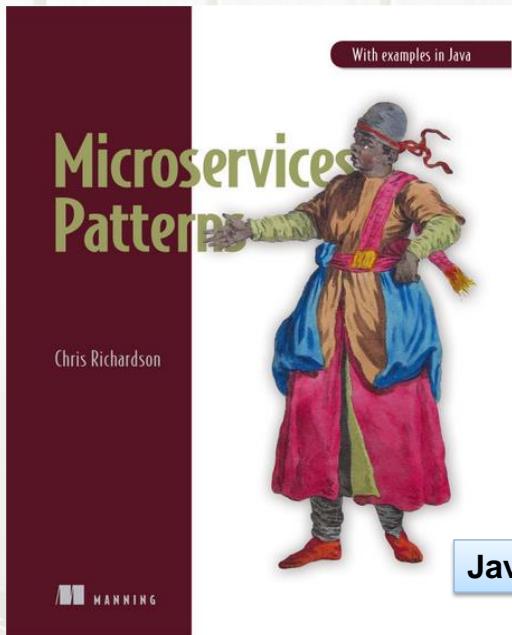
Course Summary

What Did We Learn?

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

Recommended Sources

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



**Hands-On
RESTful Python
Web Services**

Second Edition

Develop RESTful web services or APIs with modern Python 3.7



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?

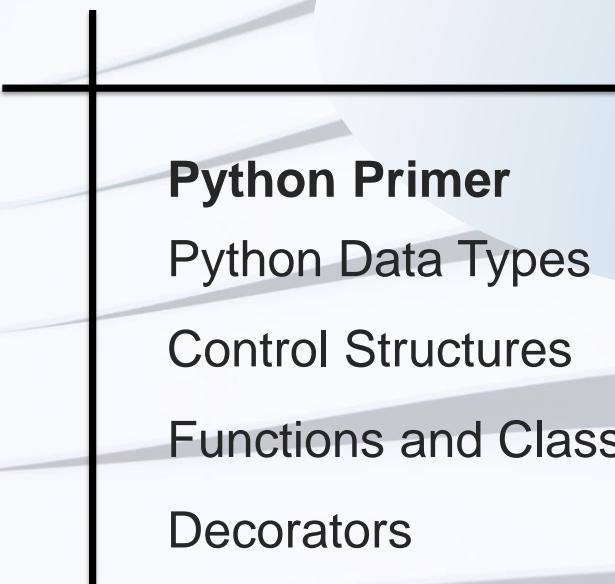


Appendix

Python Primer

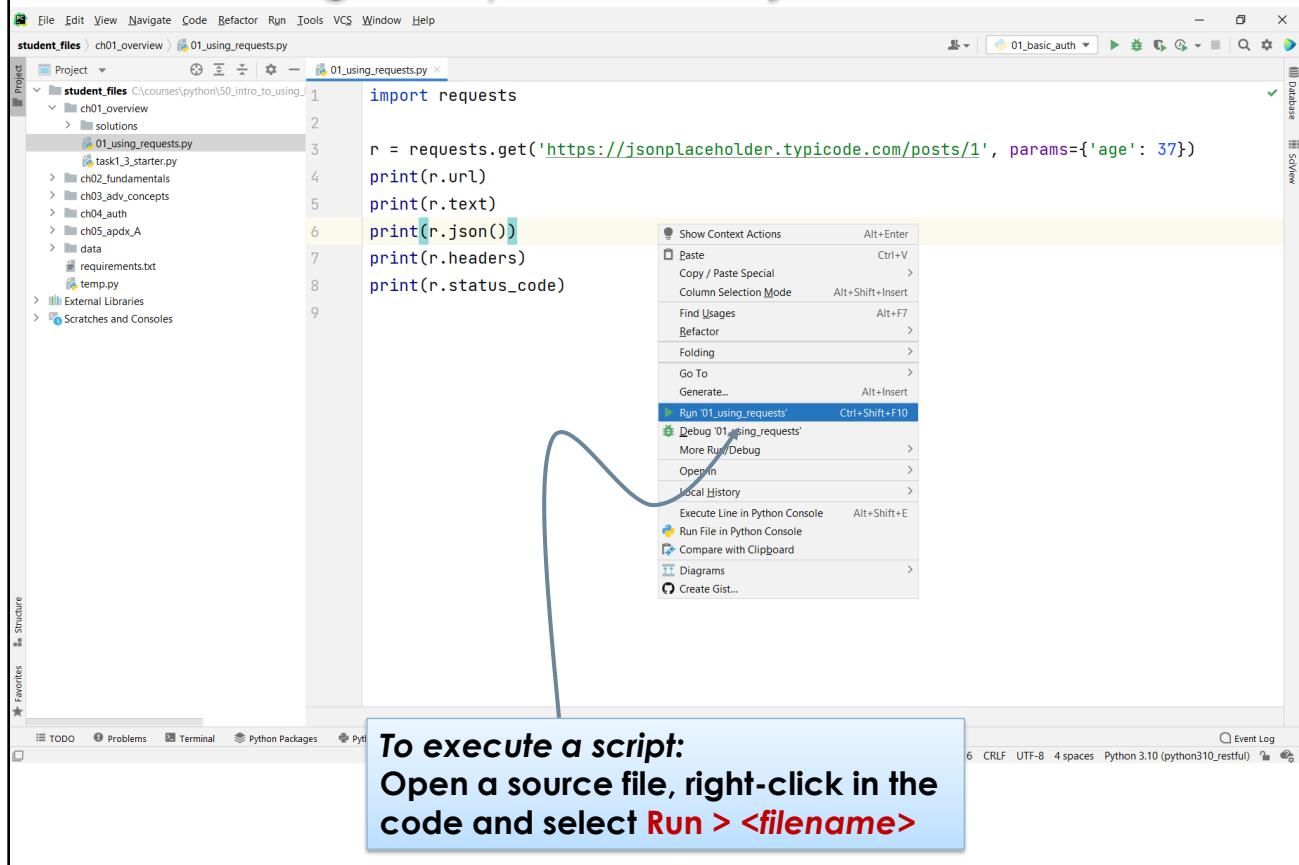


Appendix Overview



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 = []
```

Empty lists

```
my_list = list()
```

List created using the type name

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

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

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

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

```
for record in records:  
    print('{0} {1}, {2} {3}'.format(*record))
```

```
for (first, last, age, email) in records:  
    print(f'{first} {last}, {age} {email}')
```

```
for first, last, age, email in records:  
    print('{first} {last}, {age} {email}'.format(first=first,  
                                                last=last,  
                                                age=age,  
                                                email=email))
```

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**

```
d1['Smith']          # returns 43  
d1['Green']         # generates a KeyError
```

- 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:

```
d1.get('Edwards')      # returns 36  
d1.get('Green')        # returns None
```

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

```
d1.get('Cramer', 0)    # returns 29  
d1.get('Green', 0)      # returns 0
```

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'  
}
```

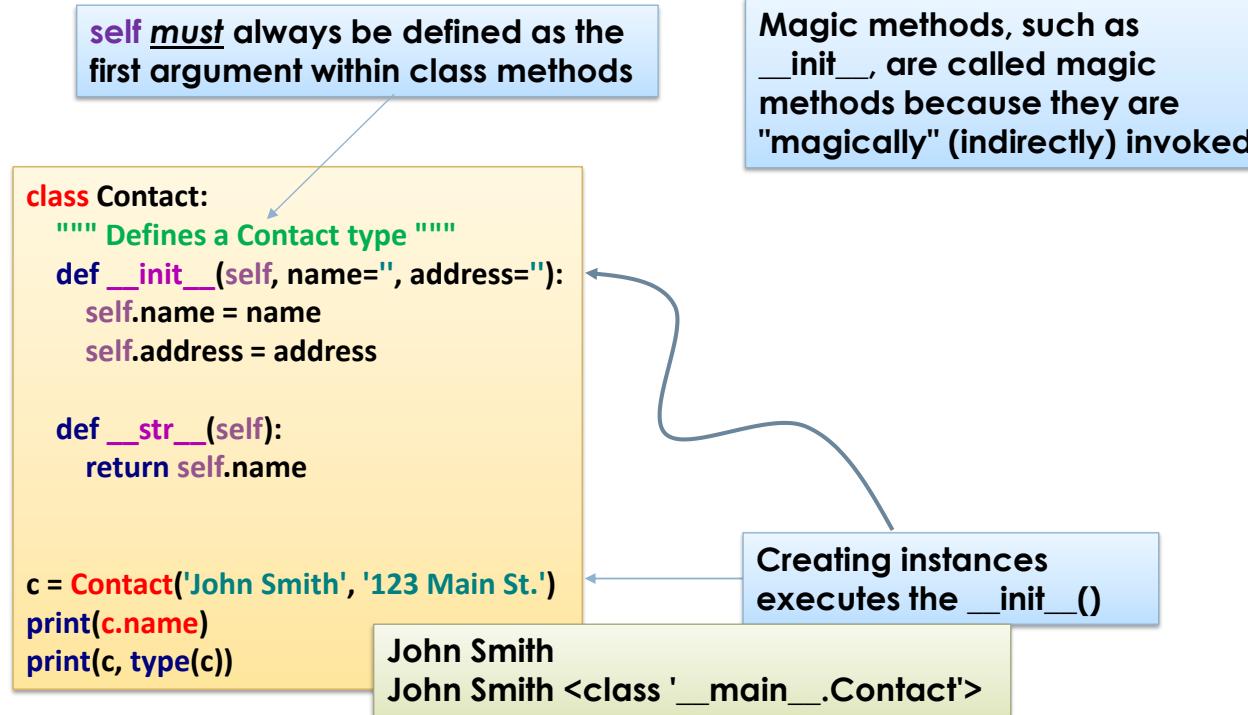
```
results = summary(cust, 1108.23)  
print(results)
```

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

FastAPI Overview



Appendix Overview

Pydantic
Intro to FastAPI

What is Pydantic?

- Pydantic is a third-party validation framework
 - Capable of performing runtime validation
 - Relies upon Python's **type hint** system

```
from pydantic import BaseModel

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

Models should inherit from
Pydantic's **BaseModel**

- Numerous frameworks in Python rely upon Pydantic for validation services: mypy, Transformers, **FastAPI!**

ch06_fastapi/01_pydantic.py

Pydantic has become a popular framework for use in everything from code quality analysis tools such as mypy to API servers such as FastAPI to natural language processing frameworks such as Transformers.

It uses Python annotations to detect desired types. By default, it will only validate when objects are instantiated.

Pydantic Validations

- Pydantic will validate objects upon instantiation

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

Pydantic models must use keyword arguments to instantiate objects

```
c = Celebrity(name='Oprah Winfrey', salary=225.0,  
               year=2005, type='Personalities')  
print(f'Celebrity: {c}')
```

Celebrity: name='Oprah Winfrey' salary=225.0 year=2005 type='Personalities'

ch06_fastapi/01_pydantic.py

The following demonstrates how to instantiate objects that come from BaseModel.

More with Instantiating Objects Using Pydantic

- Several other ways to instantiate using Pydantic

```
data = {  
    'name': 'Oprah Winfrey',  
    'salary': 225.0,  
    'year': 2005,  
    'type': 'Personalities'  
}  
c = Celebrity(**data)  
print(f'Celebrity: {c}')
```

A dictionary can be expanded into the initializer

```
c = Celebrity(name='Oprah Winfrey', salary=225.0,  
               year='2005', type='Personalities')  
print(f'Celebrity: {c}')
```

The string value is coerced into an int by Pydantic automatically

ch06_fastapi/01_pydantic.py

A dictionary can be used to expand items into a Pydantic model.

The year in the second example is a string. The class defines the year as an int. But Pydantic is able to coerce the value into an int without an error.

Errors on Validation

- When a validation error occurs, Pydantic raises a ValidationError

```
c = Celebrity(name='Oprah Winfrey', salary='225.0',
                year='some_year', type='Personalities')
```



Traceback (most recent call last):

```
  File "01_pydantic.py", line 34, in <module>
    c = Celebrity(name='Oprah Winfrey', salary='225.0', year='some year',
type='Personalities')
  File "main.py", line 176, in __init__
    self.__pydantic_validator__.validate_python(data, self_instance=self)
pydantic_core._pydantic_core.ValidationError: 1 validation error for Celebrity
year
  Input should be a valid integer, unable to parse string as an integer
[type=int_parsing, input_value='some year', input_type=str]
  For further information visit https://errors.pydantic.dev/2.7/v/int_parsing
```

ch06_fastapi/01_pydantic.py

A ValidationError is raised if a value cannot be properly validated or coerced.

Getting FastAPI Up and Running

- FastAPI looks very similar to Flask

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

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_fastapi/02_fastapi_celeb.py

FastAPI looks similar to how Flask is started. It also uses a decorator above the methods. The response_model defines what types of objects will be returned.

FastAPI and Pydantic with Celebrity

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

@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]
```

ch06_fastapi/03_fastapi_full.py

FastAPI "injects," not only the path parameters sent by the client, but also the object embedded in the body of the request.

This can be tested with an autogenerated client at <http://localhost:8000/docs>.

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
```

```
python3.12 -v
```

 (repeat this using 3.11, 3.10, or 3.9 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 what works 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 within. This way, anything we install will only go into the virtual environment and won't affect any of your current Python environments.

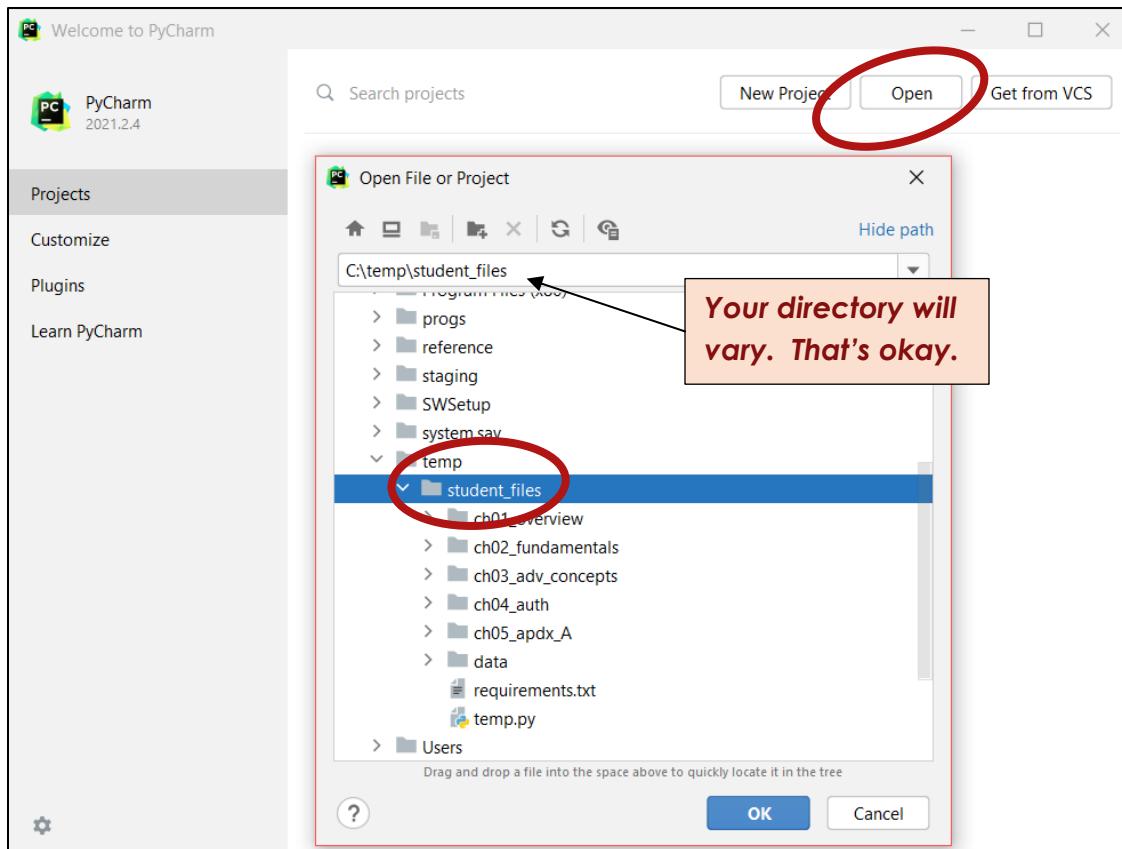
There are two ways to create the virtual environment:

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

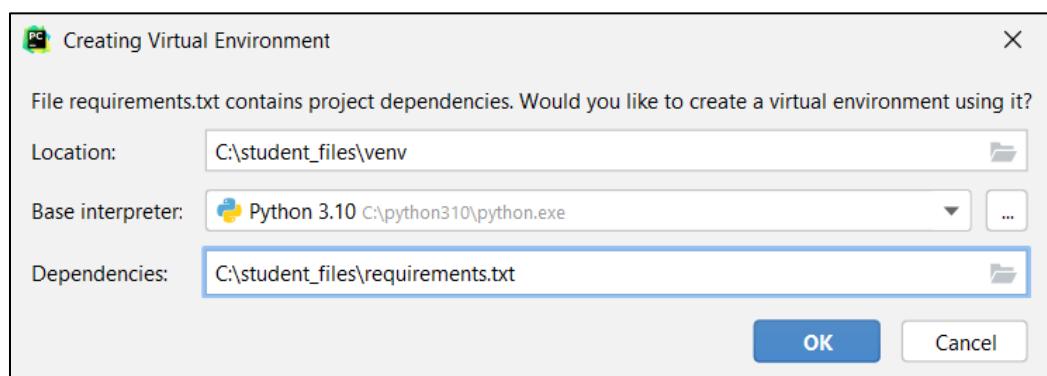
If technique #1 from above doesn't work, skip down to the star shown below and perform a manual creation of the virtual environment.

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 can 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)
```

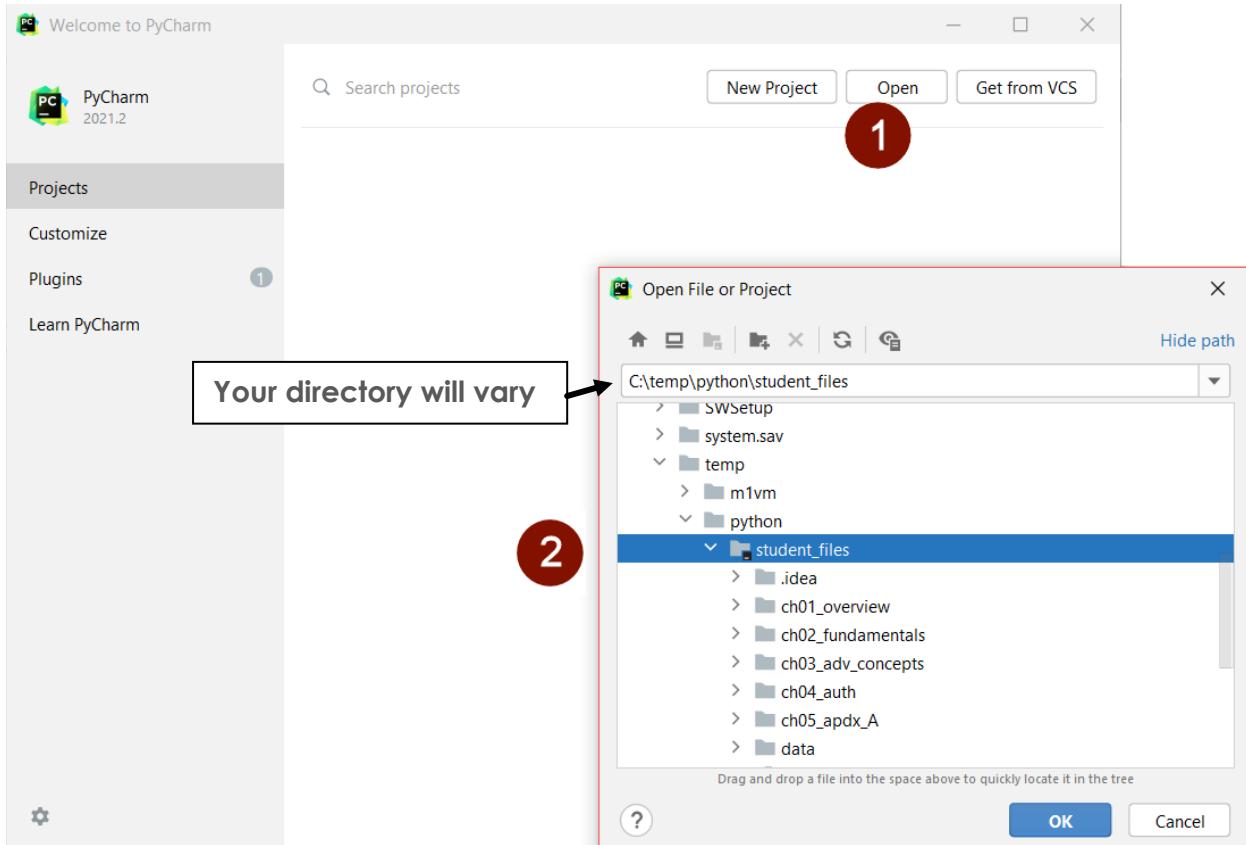


Setup the IDE

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.

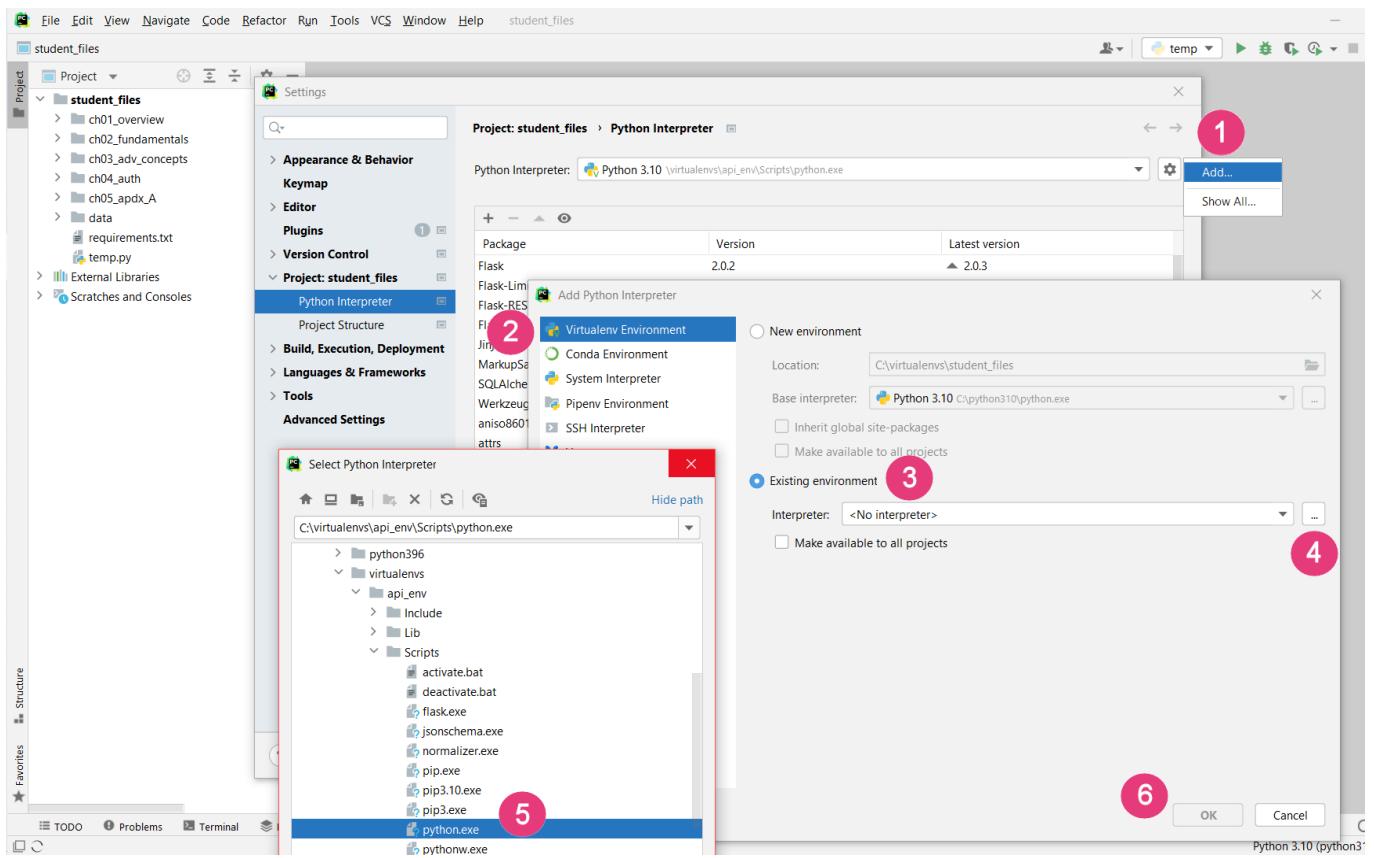


Proceed to the settings to establish which interpreter to use.

OS X: **PyCharm > Preferences**

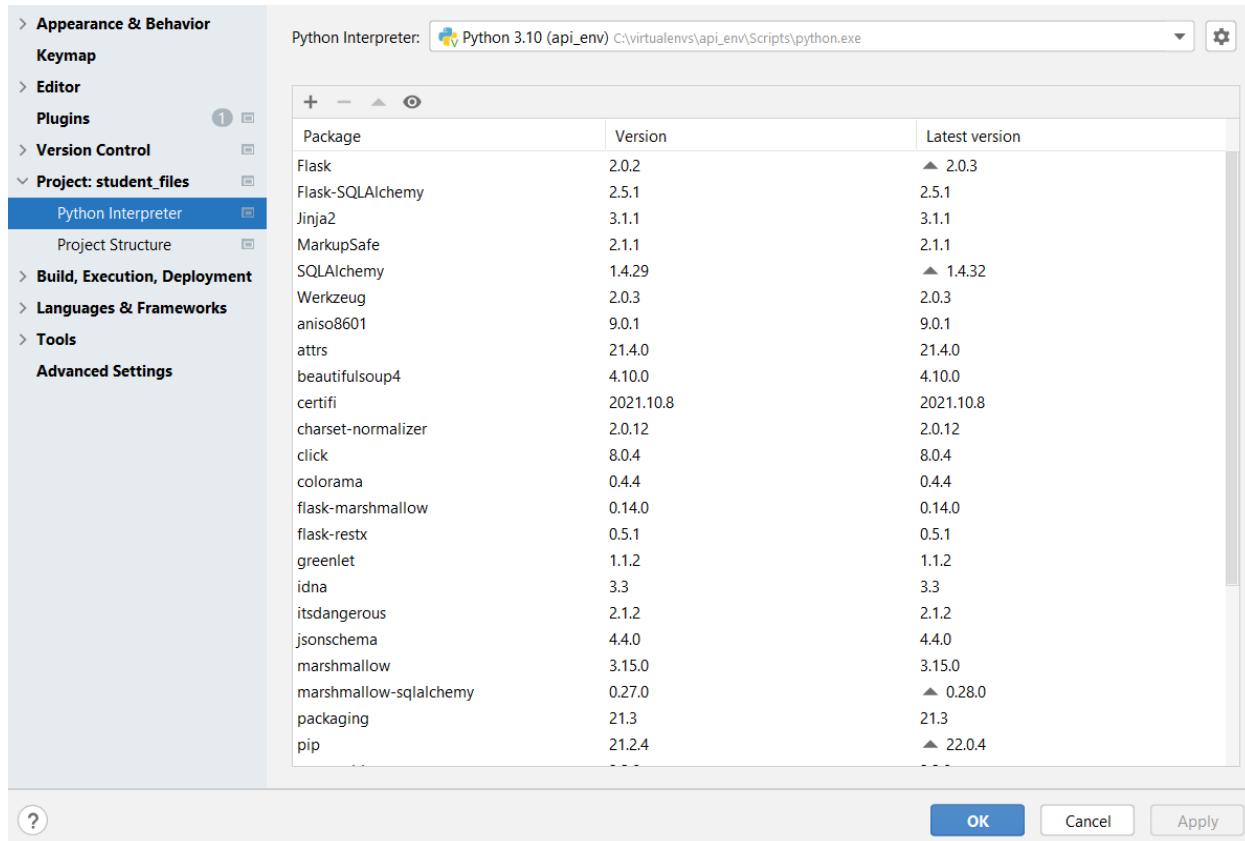
Windows: **File > Settings**

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 FastAPI server and set up our first path parameter decorator. Begin by opening the **task2_1_starter.py** file within ch02_fundamentals.



Create the FastAPI Server

At the location of step 1 in the source file, instantiate the FastAPI server object.

```
app = FastAPI()
```



Define a Route

Define a single path parameter decorator for now. It will only handle "GET" HTTP requests. It will map to:

/api/invoices/{invoice_num}. Place this decorator above the `retrieve_invoice()` function.

```
@app.get('/api/invoices/{invoice_num}')
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 here) by iterating through the list of invoices (called data) and checking for any invoice numbers (invoice_num) that match the invoice id (row[0]). Do this as follows:

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



Check for No Invoice Found

Below the statement from the previous step, check to see if any results were found. If not, respond by raising an HTTPException with a status_code of 404.

```
results = [row[1:] for row in data if invoice_num == row[0]]  
  
if not results:  
    raise HTTPException(status_code=404,  
                        detail='Invoice not found')
```



Complete the Return Statement

At the location where the return statement exists, complete it by returning the results found in step 3. Do this by providing a dictionary in the return.

```
return {'results': results}
```



Complete the uvicorn.run() Statement

Finish the task by passing the `app` object into the `uvicorn.run()` statement:

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

That's it! Test it out within your browser for now by starting the server and navigating your browser to: `http://localhost:8000/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 = click.prompt('Enter invoice number to retrieve',  
                           default=default)
```



Construct the URL to Access Our FastAPI 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 = click.prompt('Enter invoice number to retrieve',  
                           default=default)  
  
url = f'{base_url}{path}{invoice_num}'
```



3 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 = click.prompt('Enter invoice number to retrieve',  
                           default=default)  
  
url = f'{base_url}{path}{invoice_num}'  
  
results = requests.get(url).json()
```



4 Display the Retrieved Results

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

```
default = '536365'  
invoice_num = click.prompt('Enter invoice number to retrieve',  
                           default=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

Creating the POST and Other Methods



Overview

This task will implement a method within the FastAPI server which handles creating new invoices. We will not complete this method for now as it will require us to incorporate the database (shortly). You will create a Pydantic model and stub out the PUT, DELETE, and GET (all) methods as well. You'll test your solution against the provided task2_3_client.py file.



Import Needed Items

Add the import needed to create a Pydantic model. This is the **BaseModel class**.

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



Create the Invoice Model Class

Create the Invoice Model Class as defined on our working slide:

stock_code (str), quantity (int), description (str), invoice_date (str),
unit_price (float), customer_id (str), and country (str)

```
app = FastAPI()

class Invoice(BaseModel):
    stock_code: str
    quantity: int
    description: str
    invoice_date: str
    unit_price: float
    customer_id: str
    country: str
```



Create the POST Method

Create the the APIs POST handling method. Call it `create_invoice()`. Be sure to add the decorator above it. It should also accept an `Invoice` object. Place this method below the already provided `retrieve_invoice()` method.

```
@app.post('/api/invoices')
def create_invoice(invoice: Invoice):
    return {'action': 'POST response', **invoice.model_dump() }
```



Create the GET (All) Path Parameter Function

Create the GET (all) HTTP path operation (function). It should provide the path parameter decorator. Call the function `get_all_invoices()` and have it return a simple dictionary, as shown:

```
@app.get('/api/invoices')
def get_all_invoices():
    return {'action': 'GET (all) response'}
```



Create the PUT Path Parameter Function

Create the PUT HTTP path operation (function). It should provide the path parameter decorator. Call the function update_invoice() and have it return a simple dictionary, as shown:

```
@app.put('/api/invoices/{invoice_num}')
def update_invoice():
    return {'action': 'PUT response'}
```



Create the DELETE Path Parameter Function

Create the DELETE HTTP path operation (function). It should provide the path parameter decorator. Call the function delete_invoice() and have it return a simple dictionary, as shown:

```
@app.delete('/api/invoices/{invoice_num}')
def delete_invoice():
    return {'action': 'DELETE response'}
```

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.

Note: references in screenshots to port 8051 will be port 8000 instead.



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 with icons for 'Collections', 'APIs', 'Environments', 'Mock Servers', 'Monitors', 'Flows', and 'History'. The 'Collections' icon is highlighted with a red oval. In the main area, a collection named 'Invoice API' is selected, also highlighted with a red oval. The collection details show it has one request: 'GET Celebrities'. Below the collection, it says 'This collection is empty' and 'Add a request to start working.' To the right of the collection details, there are tabs for 'Authorization', 'Pre-request Script', 'Tests', and 'Variables', with 'Authorization' being the active tab. A note states: 'This authorization method will be used for every request in this collection. You can'. At the bottom, it says 'Type' and 'No Auth'.

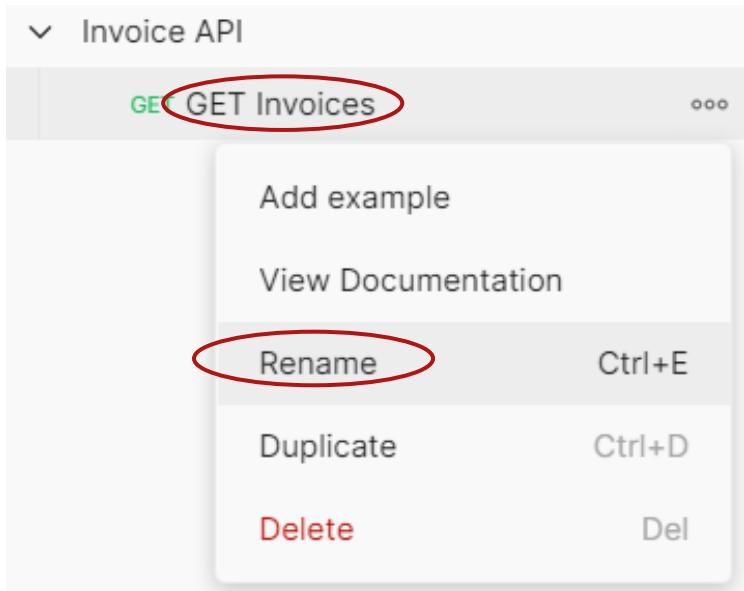


Add the Five Requests

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

The screenshot shows a context menu for the 'Invoice API' collection. The menu items are: 'Add Request' (highlighted with a red oval), 'Add Folder', 'Monitor Collection', 'Mock Collection', 'Create a fork', 'Rename' (with keyboard shortcut 'Ctrl+E'), 'Duplicate' (with keyboard shortcut 'Ctrl+D'), 'Export', and 'Delete'. The 'Add Request' option is the primary focus of this step.

Give the request a new name.



Supply the URL of `http://localhost:8000/api/invoices`.

Select the HTTP method type as shown.

A screenshot of the Postman interface showing the configuration for the "GET Invoices" request. The left sidebar shows other endpoints like "POST POST Invoices", "GET GET Invoice", "PUT PUT Invoice", and "DEL DELETE Invoice". The main panel shows the "Invoice API / GET Invoices" tab. The "Method" dropdown is set to "GET" (circled in red). The "URL" field contains the value "http://localhost:8051/api/invoices" (also circled in red).

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 the following details:

- Request Type:** POST
- URL:** http://localhost:8051/api/invoices
- Body Tab:** Selected (radio button is green)
- Body Content:** form-data
- Table Data:** (Key, Value)

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 is a sidebar with a '+' button, a filter icon, and a 'xxx' button. Below these are sections for 'Celebrity API' and 'Invoice API'. Under 'Invoice API', there are five items: 'GET GET Invoices' (highlighted in green), 'POST POST Invoices' (highlighted in orange), 'GET GET Invoice' (highlighted in grey), 'PUT PUT Invoice' (highlighted in blue), and 'DEL DELETE Invoice' (highlighted in red). A red arrow points from the 'GET GET Invoice' item in the sidebar to the 'GET' button in the main request panel. The main panel has a title 'Invoice API / GET Invoice'. To the right of the title is the URL 'http://localhost:8051/api/invoices/536365', which is circled in red. Below the title are tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', and 'Tests'. The 'Params' tab is selected. Below the tabs is a 'Query Params' section with a table:

	KEY	VALUE
	Key	Value

Below the table is a 'Response' section.

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):** A context menu is open over the "PUT" button, showing options: Open in Tab, Rename (Ctrl+E), Duplicate (Ctrl+D), and 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):** A context menu is open over the "DELETE" button, showing options: Open in Tab, Rename (Ctrl+E), Duplicate (Ctrl+D), and 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 following details:

- Collection:** Invoice API
- Request:** GET /invoices
- Method:** GET
- URL:** http://localhost:8051/api/invoices/
- Headers:** (7)
- Body:** (Raw JSON response)
- Response Status:** 200 OK (14 ms, 10.76 KB)
- Response Content (Raw):**

```
{"results100": [[{"id": "536365", "product_id": "71053", "name": "WHITE METAL LANTERN", "category": "6", "date": "12/1/2010 8:26", "price": "3.39", "country": "United Kingdom"}, {"id": "536365", "product_id": "84466B", "name": "CREAM CUPID HEARTS COAT HANGER", "category": "8", "date": "12/1/2010 8:26", "price": "2.75", "country": "United Kingdom"}, {"id": "536365", "product_id": "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_id": "84029E", "name": "RED WOOLLY HOTTIE WHITE HEART.", "category": "6", "date": "12/1/2010 8:26", "price": "3.39", "country": "United Kingdom"}, {"id": "536365", "product_id": "22752", "name": "SET 7 BABUSHKA NESTING BOXES", "category": "2", "date": "12/1/2010 8:26", "price": "7.65", "country": "United Kingdom"}, {"id": "536365", "product_id": "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_id": "226331", "name": "HAND WARMER UNION JACK", "category": "6", "date": "12/1/2010 8:28", "price": "1.85", "country": "United Kingdom"}, {"id": "536366", "product_id": "226321", "name": "HAND WARMER RED POLKA DOT", "category": "6", "date": "12/1/2010 8:28", "price": null, "country": null}]]}
```

Task 2-5

Adding SQLAlchemy, Implementing POST and Get (Singular)



Overview

This task incorporates the database SQLAlchemy. You will add SQLAlchemy into our on-going invoice API, create a database model object, write and test the `create_invoice()` (POST) method and the `retrieve_invoice()` (GET) method.



Create the Session Class

Create the SQLAlchemy Session class as shown:

```
engine = create_engine('sqlite:///+' + str(db_url), echo=True)
Session = sessionmaker(bind=engine)
session = Session()
```



Create the Base Class

Create the SQLAlchemy Base class using `declarative_base()`

```
Session = sessionmaker(bind=engine)
session = Session()

Base = declarative_base()
```



Build the InvoiceModel Class

In the blank line provided, create the InvoiceModel class that inherits from Base. Add the following in the blank space provided:

```
class InvoiceModel(Base):  
    __tablename__ = 'purchases'
```

Uncomment the lines of code that provide the Column definitions. Your class to this point should look like the following:

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



Continue Building InvoiceModel

Complete the `__init__()` of the InvoiceModel class. Do this by adding the following line and uncommenting the `self.xxx = xxx` lines as shown:

```
def __init__(self, invoice_num, stock_code, quantity,
            description, invoice_date,
            unit_price, customer_id, country):
```

The final version of our invoice model should look like this:

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

    def __init__(self, invoice_num, stock_code, quantity,
                 description, invoice_date,
                 unit_price, customer_id, country):
        super().__init__()
        self.invoice_num = invoice_num
        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_num} {self.stock_code} Qty:\
{self.quantity} - {self.description}'
```

```
def to_dict(self) -> dict:  
    return {'id': self.id, 'invoice_num': self.invoice_num,  
            'stock_code': self.stock_code,  
            'quantity': self.quantity,  
            'description': self.description,  
            'invoice_date': self.invoice_date,  
            'unit_price': self.unit_price,  
            'customer_id': self.customer_id,  
            'country': self.country}  
  
__repr__ = __str__
```



Complete the retrieve_invoice() Method

Remove the pass statement from retrieve_invoice() and implement the following within it:

```
invoice = session.get(InvoiceModel, id)  
if not invoice:  
    raise HTTPException(status_code=404,  
                        detail=f'Invoice not found.')  
return invoice.to_dict()
```

The final version of retrieve_invoice() should look like the following:

```

@app.post('/api/invoices')
def create_invoice(invoice_request: Invoice):
    try:
        new_invoice_db =
            InvoiceModel(**invoice_request.model_dump())
        session.add(new_invoice_db)
        session.commit()
        results = new_invoice_db.to_dict()
    except Exception as err:
        raise HTTPException(status_code=404,
                            detail=f'Error creating invoice: {err.args[0]}')

    return results

```



Complete the create_invoice() Method

Remove the pass statement from create_invoice(). Add the following to the try block.

```

try:
    new_invoice_db =
        InvoiceModel(**invoice_request.model_dump())
    session.add(new_invoice_db)
    session.commit()
    results = new_invoice_db.to_dict()
except Exception as err:
    raise HTTPException(status_code=404,
                        detail=f'Error creating invoice: {err.args[0]}')

return results

```

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