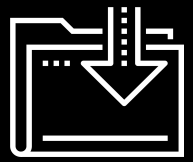
Introduction to Flask & Serving Data with APIs

Data Boot Camp

Lesson 10.3



Class Objectives

By the end of today's class you will be able to:

- Use SQLAlchemy ORM to model tables.
- Perform CRUD with SQLAlchemy.
- Reflect existing databases with SQLAlchemy.
- Plot query results from SQLAlchemy ORM.

Joins & Dates

Joins in SQLAlchemy are very similar to joins in Pandas!

SQLAlchemy Joins

SQLAlchemy joining tables step-by-step:

01 Use `inspect(engine).get_table_names()` to find table names in the database

02 Use `inspect(engine).get_columns(table)` to get the column names

03 Create a list of all table columns you wish to keep

04 Use `.filter()` to describe what columns to join on

Instructor Demonstration

SQLAlchemy Joins

SQLAlchemy Dates

Times and dates are bit trickier than integers or decimals:

01

In some cases we may need to do conversions to add or subtract time:

- Days, months, years to seconds
- Then convert everything back!

02

Many ways to annotate a date:

- 10/21/2020
- 21/10/2020
- 21Oct2020
- October 21, 2020

03

Python libraries like `datetime` makes things easier!

Datetime

Datetime and SQLAlchemy work well together!

Dates and times can be stored in many ways:

- Datetime objects
- Strings
- Integers (number of seconds)

```
# Query for the Dow closing price for `CSCO`
# 1 week before `2011-04-08` using the datetime library
query_date = dt.date(2011, 4, 8) - dt.timedelta(days=7)
print("Query Date: ", query_date)
```

```
Query Date:  2011-04-01
```

```
session.query(Dow.date, Dow.close_price).\
    filter(Dow.stock == 'CSCO').\
    filter(Dow.date == query_date).all()
```

```
[('2011-04-01', 17.04)]
```

It could be difficult to compare, or query for a specific date/time Python's datetime library helps make dates and times easier

Instructor Demonstration

Datetime

Activity: Dates

In this activity, you will practice working with dates, both in SQLAlchemy and with the `datetime` library.

(Instructions sent via Slack.)

Suggested Time:

15 minutes

Activity: Dates

Instructions	Use the provided `dow.sqlite` dataset to analyze the average stock prices (average open, average high, average low, and average close) for all stocks in the month of May.
	Plot the results as a Pandas or Matplotlib bar chart.
Bonus	Calculate the high-low peak-to-peak (PTP) values for `IBM` stock after `2011-05-31`. The high-low PTP is calculated by subtracting `low_price` from `high_price`.
	Use a `DateTime.date` object in the query filter.
	Use list comprehension to create a list of dictionaries from the query results.
	Create a DataFrame from the list of dictionaries.
	Use the `boxplot()` method on the DataFrame to plot PTP distribution statistics.

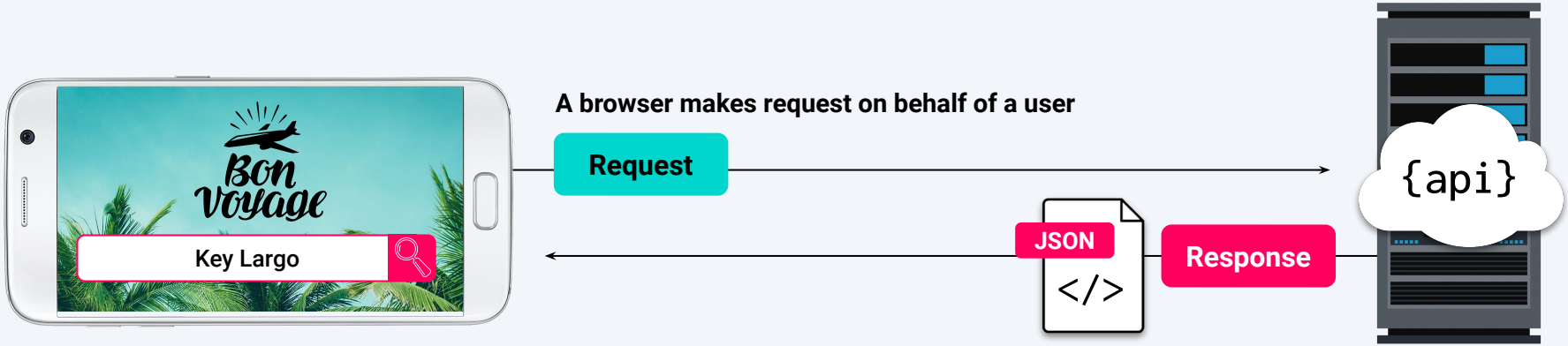
Time’s Up! Let’s Review.

Introduction to Flask

Internet is Built from Clients and Servers

A server is essentially a program

- We can write the code that runs a server
- We can determine what data is displayed
- We can determine what data is shared

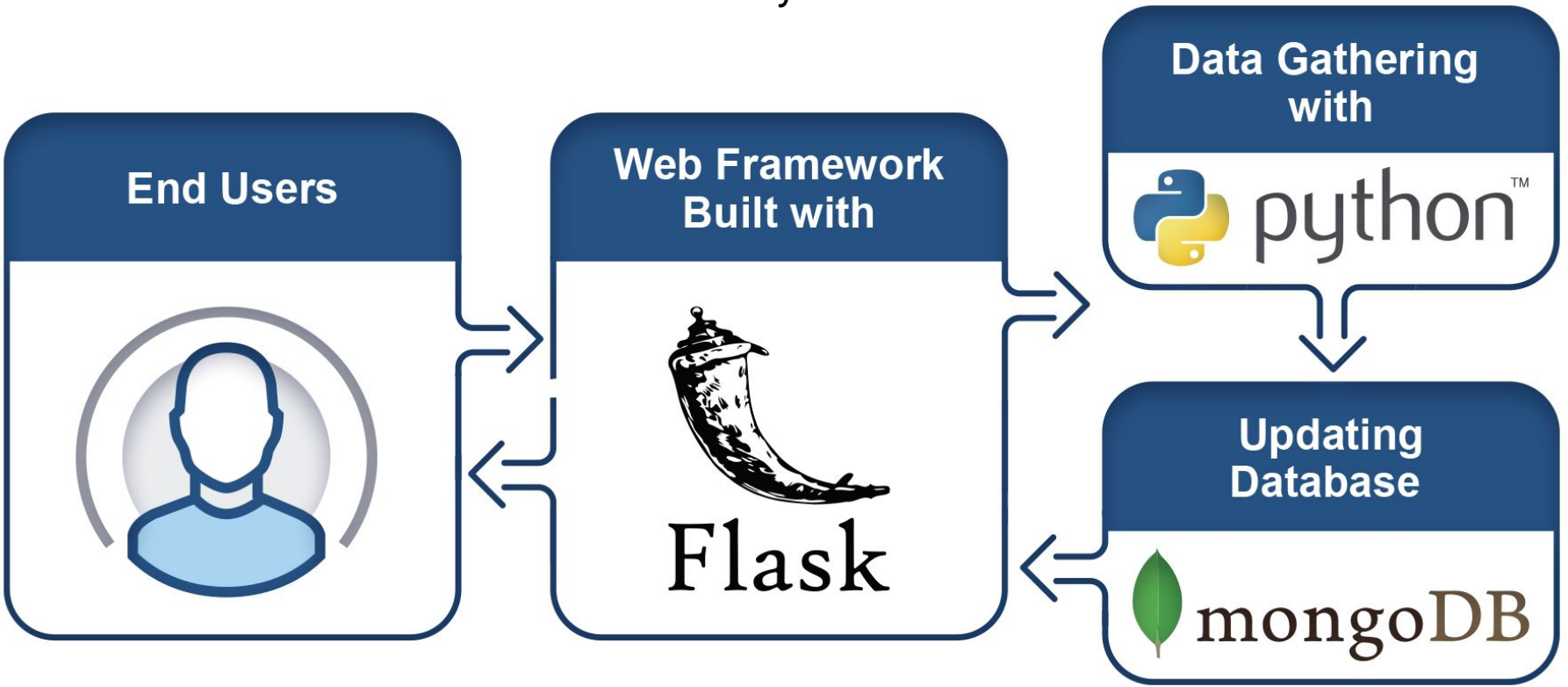


Whatever application or device that is asking for information is called a “client”

A “server” is a process running on a remote machine listening for requests.

Introduction to Flask

Flask is a micro web framework to build your own APIs!



Instructor Demonstration

Steps with Flask

Activity: Hello, Web

In this activity, you will create your first Flask server with a few endpoints.

(Instructions sent via Slack.)

Suggested Time:

10 minutes

Activity: Hello, Web

Instructions	Create an `app.py`, and make the necessary imports.
	Use Flask to create an `app` instance.
	Use route decorators to define the following endpoints: • `/`, or your **index route**: This should return a simple string, such as `"Hello, world!"` or `"Welcome to my API!"` • `/about`, which should return a string containing your name and current location • `/contact`, which should return a string telling visitors where to email you
	Finally, add code at the end of the file that will allow you to run the server from the command line with python `app.py`.
Hint	Refer to the Flask documentation as you work through this activity.

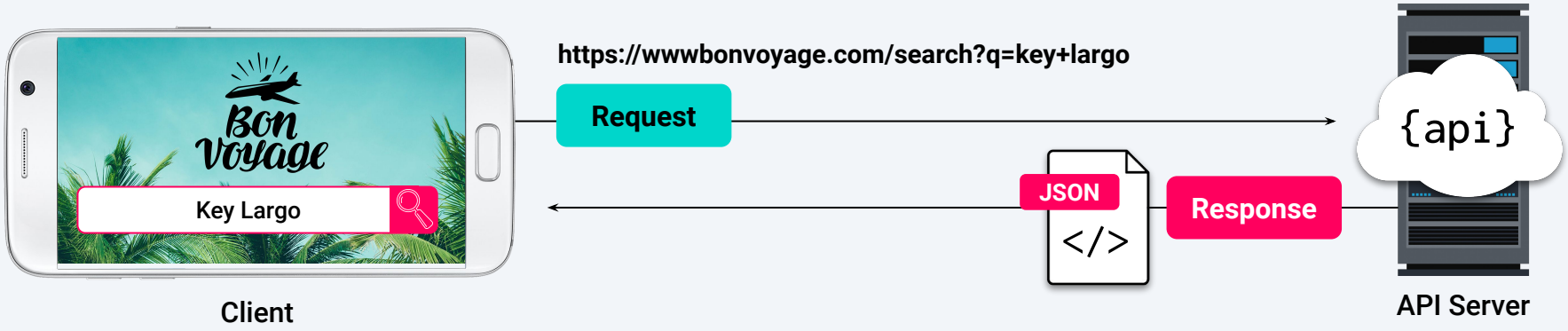
Time’s Up! Let’s Review.

JSON APIs with jsonify

JSON APIs with jsonify

All of the routes that we've written so far have returned ***string*** responses.

The APIs we've dealt with do not return raw text; rather, they return JSON data.



JSON APIs with jsonify

Flask has a function to create JSON responses

We cannot simply return a dictionary response directly through Python.
Fortunately, Python dictionaries map naturally to JSON.

`jsonify` automatically converts Python dictionaries into JSON responses

The converted JSON responses are wrapped in HTTP to send back to the client

```python
from flask import Flask, jsonify


app = Flask(__name__)

hello_dict = {"Hello": "World!"}


@app.route("/")
def home():
    return "Hi"


@app.route("/normal")
def normal():
    return hello_dict


@app.route("/jsonified")
def jsonified():
    return jsonify(hello_dict)
```

Instructor Demonstration

JSON APIs with jsonify

Activity: Justice League

In this activity, you will create a server that sends welcome text at one endpoint, and JSON data at another endpoint.

(Instructions sent via Slack.)

Suggested Time:

20 minutes

Activity: Justice League

Instructions	
	Create a file called `app.py` for your Flask app.
	Define a Python dictionary containing the superhero name and real name for each member of the Justice League (Superman, Batman, Wonder Woman, Green Lantern, Flash, Aquaman, and Cyborg). • You can gather that information from the Justice League Wikipedia page. • Only gather the information for the 7 characters just listed.
	Create a **get** route called `/api/v1.0/justice-league`. • Inside of your **get** route, create a function called `justice_league` that will use `jsonify` to convert the dictionary of Justice League members to a JSON object and return that data as a request.
	Define a root route, `/`, that will return the usage statement for your API.

Time’s Up! Let’s Review.


Break

Routes with Variable Paths

Our current API is one-dimensional

Our current API can only return the ***entire*** Justice League dataset.

```
@app.route("/api/v1.0/justice-league")
 def justice_league():
     """Return the justice league data as json"""

      return jsonify(justice_league_members)
```

Our current API is one-dimensional

Ideally, clients can send a request for a character and expect either:

01 A JSON response with only specific character information; or

02 A detailed error response

```
return {"error": f"Character with real_name
{real_name} not found."}, 404
```

Instructor Demonstration

Routes with Variable Paths

Activity: Routes with Variable Rules

In this activity, you will add an additional API route that returns a JSON containing an individual superheroes information.

(Instructions sent via Slack.)

Suggested Time:

20 minutes

Activity: Routes with Variable Rules

Instructions

Using the previous activity as a starting point, add code to get a specific hero's information based on their superhero name.

Time’s Up! Let’s Review.

Flask with ORM

It is time to put all of the pieces together!

Flask with ORM

A useful API will enable the client to make requests and queries on massive datasets. Potentially too large to load into memory.

SQLAlchemy can be used to perform queries based on a flask route

Convert the query into a dictionary, then into a JSON with `jsonify`

Return the JSON query to the endpoint

Instructor Demonstration

Flask with ORM

Activity: Chinook Database Analysis

In this activity, you will practice analyzing databases using the SQLAlchemy ORM.

(Instructions sent via Slack.)

Suggested Time:

20 minutes

Activity: Chinook Database Analysis

Instructions

- Create a Jupyter notebook for your analysis.
- Create a SQLAlchemy engine to the database chinook.sqlite.
- Use `automap_base` to reflect the database tables.
- Create references to the `invoices` and `invoice_items` tables, and call them `Invoices` and `Items`, respectively.
- Create a SQLAlchemy ORM session object.
- Design a query that lists all of the billing countries found in the invoices table.

Design a query that lists the invoice totals for each billing country, and sort the output in descending order.

- The results tuple should contain the country name and the invoice total for that country, using all records in the invoices table.

Design a query that lists all of the billing postal codes for the USA.

Calculate the item totals, using **sum(UnitPrice * Quantity)**, for the USA.

- Return the value as a scalar floating-point number.

Calculate the invoice item totals using **sum(UnitPrice * Quantity)** for each billing postal code in the USA.

Time’s Up! Let’s Review.

Questions?