Creating APIs

Creating your own API

- Being able to convert a dataset into an API also makes it possi
  create your own custom APIs, whether that be for in-house us
  share with end-users. It lets you interact with your raw data in
  hands-on manner.
- If you have data you wish to share, an API is one way to do thi
  However, APIs are not always the best way of sharing data wit
  If the size of the data you are providing is relatively small, you
  instead provide a “data dump” in the form of a downloadable
  CSV, or SQLite file.
- You could provide both a data dump and an API, and individua
  may use one or the other to best match their needs.

Flask

- Flask is a Python library that provides a framework for web development. It is great for rapid development as it comes wit simple-yet-extensible core functionality.

- We will see an example of how to send a JSON-like response u Flask.

Command line

- It is not advised to run Flask applications through Jupyter Note or Spyder.
- We will write the code for the apps in Jupyter Notebook or Sp
- We will run the app scripts from the command line (CMD.exe in Anaconda).
- To do this we need to know some basic commands for creatin folders, navigating to folders, and running Python scripts from command line.

Commands

- **dir** to see all files or folders in the directory you are currently i
  Linux).
- **cd** to change the directory eg **cd DKIT** to move into the DKIT fo
- **cd ..** to go out of the current folder by one step.
- **mkdir my_folder** to make a new folder called my_folder in the
  current directory.
- **Python my_script.py** to run the script called my_script.py.
- Use the tab button on your keyboard to autofill words when t

A simple Flask application

- We will write the code for a simple Flask app in Jupyter Noteb

- Save the code from the following slide in a script called app.py
  known folder.

A simple Flask application

```
import flask

app = flask.Flask(__name__)
app.config["DEBUG"] = True

@app.route('/', methods=['GET'])
def home():
    return "<h1>Distant Reading Archive</h1><p>This site is a prototype AP
distant reading of science fiction novels.</p>"

app.run()
```

Running the script from the command lin

- Use cd to navigate to the folder containing the app.py script in
  command line.
- Use dir to check that the script is in the folder.
- When in the correct folder, use the command **Python app.py**
  the script app.py.
- The output will tell you ‘Running on http://127.0.0.1:5000/ (P
  CTRL+C to quit)’.
- Go to the link http://127.0.0.1:5000/ to see the working web
- Sometimes CTRL+C does not kill the app. I use CTRL+ScrLk to k
  that case.

What is the Flask code doing?

- Flask maps HTTP requests to Python functions. In this case, w
  mapped one URL path (‘/’) to one function, home. When we c
  to the Flask server at http://127.0.0.1:5000/, Flask checks if th
  match between the path provided and a defined function. Sin
  been mapped to the home function, Flask runs the code in the
  function and displays the returned result in the browser. In thi
  the returned result is HTML markup for a home page welcomi
  visitors to the site hosting our API.
- The process of mapping URLs to functions is called **routing**.

What is the Flask code doing?

- The @app.route('/', methods=['GET']) syntax is the part of the program t Flask know that this function, home, should be mapped to the path /. Th methods list (methods=['GET']) is a keyword argument that lets Flask kn kind of HTTP requests are allowed. We'll only be using GET requests, but web applications need to use both GET requests (to send data from the application to the user) and POST requests (to receive data from a user).
- import flask — Imports the Flask library
- app = flask.Flask(__name__) — Creates the Flask application object, whi contains data about the application and also methods (object functions) the application to do certain actions. app.config and app.run() are such m
- app.config["DEBUG"] = True — Starts the debugger. With this line, if you contains mistakes, you'll see an error when you visit your app. Otherwise only see a generic message in the browser when there's a problem with code.
- app.run() — A method that runs the application server.

Publishing data as an API

- We will now write a similar script called app2.py.
- We will create a list of dictionaries called books.
- After creating the home page, we create a route to return all t available entries in our dataset.
- The dataset is converted into a JSON file using jsonify before publishing to the route.

The data

```
books = [
   {'id': 0,
    'title': 'A Fire Upon the Deep',
    'author': 'Vernor Vinge',
    'first_sentence': 'The coldsleep itself was dreamless.',
    'year_published': '1992'},
   {'id': 1,
    'title': 'The Ones Who Walk Away From Omelas',
    'author': 'Ursula K. Le Guin',
    'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of Summer came to the city Omel
towered by the sea.',
    'published': '1973'},
   {'id': 2,
    'title': 'Dhalgren',
    'author': 'Samuel R. Delany',
    'first_sentence': 'to wound the autumnal city.',
    'published': '1975'}
]
```

```python
import flask
from flask import request, jsonify

app = flask.Flask(__name__)
app.config["DEBUG"] = True

# Create some test data for our catalog in the form of a list of dictionaries.
books = [
    {'id': 0,
     'title': 'A Fire Upon the Deep',
     'author': 'Vernor Vinge',
     'first_sentence': 'The coldsleep itself was dreamless.',
     'year_published': '1992'},
    {'id': 1,
     'title': 'The Ones Who Walk Away From Omelas',
     'author': 'Ursula K. Le Guin',
     'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of Summer came to the city Omelas, brig
     'published': '1973'},
    {'id': 2,
     'title': 'Dhalgren',
     'author': 'Samuel R. Delany',
     'first_sentence': 'to wound the autumnal city.',
     'published': '1975'}
]


@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''


# A route to return all of the available entries in our catalog.
@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)

app.run()
```

Viewing the data on the API

- Run the script similarly to before. Once the server is running, v
  route URL to view the data:
  http://127.0.0.1:5000/api/v1/resources/books/all

- We see JSON output for the three entries in our test catalog.

- At this point, we've created a working, if limited, API. In the ne
  section, we'll allow users to find books via more specific data,
  an entry's ID.

Allowing filtering capability

- We will add a function that allows users to filter their returned resu
  using a more specific request.
- Run the script app3.py similarly to before. Once the server is runni
  the following URLs to see the filtering in action:
  127.0.0.1:5000/api/v1/resources/books?id=0
  127.0.0.1:5000/api/v1/resources/books?id=1
  127.0.0.1:5000/api/v1/resources/books?id=2
  127.0.0.1:5000/api/v1/resources/books?id=3
- Each of these returns a different entry, except for the last, which sh
  return an empty list: [], since there is no book for which the id valu

New code to allow filtering capability

```
@app.route('/api/v1/resources/books', methods=['GET'])
def api_id():
    # Check if an ID was provided as part of the URL.
    # If ID is provided, assign it to a variable.
    # If no ID is provided, display an error in the browser.
    if 'id' in request.args:
        id = int(request.args['id'])
    else:
        return "Error: No id field provided. Please specify an id."

    # Create an empty list for our results
    results = []

    # Loop through the data and match results that fit the requested ID.
    # IDs are unique, but other fields might return many results
    for book in books:
        if book['id'] == id:
            results.append(book)

    # Use the jsonify function from Flask to convert our list of
    # Python dictionaries to the JSON format.
    return jsonify(results)
```

New code to allow filtering capability

- In this code, we first create a new function, called api_id, with
  @app.route syntax that maps the function to the path
  /api/v1/resources/books. That means that this function will ru
  we access http://127.0.0.1:5000/api/v1/resources/books. (No
  accessing this link without providing an ID will give the error m
  we provided in the code: Error: No id field provided. Please sp
  id.)

New code to allow filtering capability

We do two things in our function:

- First, examine the provided URL for an id and select the books that
  that id. The id must be provided like this: ?id=0. Data passed throu
  like this (after the ?) are called query parameters—we've seen ther
  when we downloaded data from APIs.
- This part of the code determines if there is a query parameter, like
  and then assigns the provided ID to a variable:

```
if 'id' in request.args:
    id = int(request.args['id'])
else:
    return "Error: No id field provided. Please specify an id."
```

New code to allow filtering capability

The second thing we do in our function is to move through our
catalog of books, match those books that have the provided ID,
append them to the list that will be returned to the user:

```
for book in books:
    if book['id'] == id:
        results.append(book)
```

Finally, the return jsonify(results) line takes the list of results an
renders them in the browser as JSON.

Designing APIs

- Our next version of our API will pull in data from a database b
  providing it to a user. It will also take additional query parame
  allowing users to filter by fields other than ID.
- Two aspects of a good API are usability and maintainability. W
  keep this in mind going forward.
- We will be designing a REST API.

Designing APIs

- The prevailing design philosophy of modern APIs is called REST.
- REST is based on four methods defined by the HTTP protocol: POST
  PUT, and DELETE. These correspond to the four traditional actions
  performed on data in a database: CREATE, READ, UPDATE, and DEL
  will only use GET requests, which correspond to reading from a dat
- Because HTTP requests are so integral to using a REST API, many de
  principles revolve around how requests should be formatted. We h
  already created one HTTP request, which returns all books provide
  sample data. To understand the considerations that go into formatt
  request, we examine a weak or poorly-designed example of an API
  endpoint.

http://api.example.com/getbook/10

The formatting of this request has a number of issues:

1. In a REST API, our verbs are typically GET, POST, PUT, or DELE
   are determined by the request method rather than in the re
   URL. That means that the word “get” should not appear in o
   request, since “get” is implied by the fact that we’re using a
   GET method.
2. Resource collections such as books or users should be denot
   plural nouns. This makes it clear when an API is referring to
   collection (books) or an entry (book).

Improving on the last slide

Incorporating the ideas from the last slide, our API would look like th

http://api.example.com/books/10

The above request uses part of the path (/10) to provide the ID. Whi
not an uncommon approach, it's inflexible — with URLs constructed
manner, you can generally only filter by one field at a time. Query
parameters allow for filtering by multiple database fields and make r
sense when providing "optional" data, such as an output format:

http://api.example.com/books?author=Ursula+K.+Le
Guin&published=1969&output=xml

Improving your API

When designing how requests to your API should be structured, makes sense to plan for future additions. Even if the current ver your API serves information on only one type of resource—book example—it makes sense to plan as if you might add other reso non-resource functionality to your API in the future:

http://api.example.com/resources/books?id=10

Improving your API

Adding an extra segment on your path such as “resources” or “e
gives you the option to allow users to search across all resource
available, making it easier for you to later support requests such
these:

https://api.example.com/v1/resources/images?id=10

https://api.example.com/v1/resources/all

Improving your API

Another way to plan for your API's future is to add a version number
path. This means that, should you have to redesign your API, you car
continue to support the old version of the API under the old version
while releasing, for example, a second version (v2) with improved or
different functionality. This way, applications and scripts built using t
version of your API won't cease to function after your upgrade.

After incorporating these design improvements, a request to our API
look like this:

https://api.example.com/v1/resources/books?id=10

API documentation and examples

- Your API should have documentation describing the resources
  functionality available through your API that also provides cor
  working examples of request URLs or code for your API. You sh
  have a section for each resource that describes which fields, s
  id or title, it accepts. Each section should have an example in t
  of a sample HTTP request or block of code.

Database example

- This last example of our Distant Reading Archive API pulls in da a database, implements error handling, and can filter books by publication date. The database used is SQLite, a lightweight da engine that is supported in Python by default. SQLite files typi end with the .db file extension.

- Before we modify our code, first download the example datab from Moodle and copy the file to your folder containing the A scripts. The final version of our API will query this database wh returning results to users.

Database example

- Run the database_app.py script from the command line.
- Once this example is running, try out the filtering functionality these HTTP requests:

http://127.0.0.1:5000/api/v1/resources/books/all

http://127.0.0.1:5000/api/v1/resources/books?author=Connie+

http://127.0.0.1:5000/api/v1/resources/books?author=Connie+published=1993

http://127.0.0.1:5000/api/v1/resources/books?published=2010

Database example

- The database downloaded for this lesson has 67 entries, one for each of winners of the Hugo Award for best science fiction novel between 1953 2014. Our API allows users to filter by three fields: id, published (year of publication), and author.
- http://127.0.0.1:5000/api/v1/resources/books/all returns all entries in t database.
- http://127.0.0.1:5000/api/v1/resources/books?author=Connie+Willis re books by the author Connie Willis. Note that, within a query parameter, between words are denoted with a + sign, hence Connie+Willis.
- http://127.0.0.1:5000/api/v1/resources/books?author=Connie+Willis&p =1993 filters by two fields—author and year of publication. Instead of th books returned by requesting ?author=Connie+Willis, this request returr the entry to The Doomsday Book, published in 1993.
- http://127.0.0.1:5000/api/v1/resources/books?published=2010 returns winners from the year 2010 (note that, in some years, more than one Hu awarded).

Database example

- When our user requests an entry or set of entries, our API pul
  information from the database by building and executing an S
  query.

Understanding the code

Relational databases allow for the storage and retrieval of data,
is stored in tables. Tables are similar to spread sheets in that the
columns and rows—columns indicate what the data represents,
“title” or “date.” Rows represent individual entries, which could
books, users, transactions, or any other kind of entity.

The database we’re working with has five columns: id, publishe
author, title, and first_sentence. Each row represents one book
won the Hugo award in the year under the published heading, a
text of which begins with the sentence in the first_sentence col

Understanding the code

The api_all function pulls in data from our Hugo database:

```
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM books;').fetchall()

    return jsonify(all_books)
```

- First, we connect to the database using our sqlite3 library: the conn variable.
- The conn.row_factory = dict_factory line lets the connection object know to use the dict_factory we've defined, which returns items from the database as dictionaries rather than lists—these wo when we output them to JSON.
- We then create a cursor object (cur = conn.cursor()), which is the object that actually moves thro database to pull our data.
- Finally, we execute an SQL query with the cur.execute method to pull out all available data (*) fr books table of our database.
- At the end of our function, this data is returned as JSON: jsonify(all_books).

Understanding the code

The purpose of our page_not_found function is to create an error pa
by the user if the user encounters an error or inputs a route that has
defined:

```
@app.errorhandler(404)
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not be found.</p>",
```

In HTML responses, the code 200 means “OK”(the expected data
transferred), while the code 404 means “Not Found” (there was no r
available at the URL given). This function allows us to return 404 pag
something goes wrong in the application.

Understanding the code

api_filter is an improvement on the api_id function that returns a book based on
new function allows for filtering by id, published, and author. The function first gr
query parameters provided in the URL (remember, query parameters are the part
URL that follows the ?, like ?id=10).

query_parameters = request.args

It then pulls the supported parameters id, published, and author and binds them
appropriate variables:

id = query_parameters.get('id')

published = query_parameters.get('published')

author = query_parameters.get('author')

Understanding the code

The next segment writes an SQL query that will be used to find the requested inf
in the database. SQL queries used to find data in a database take this form:

SELECT <columns> FROM <table> WHERE <column=match> AND <column=match

To get the correct data, we need to build both an SQL query that looks like the ab
list with the filters that will be matched. Combined, the query and the filters prov
the user will allow us to pull the correct books from our database.

We begin to define both the query and the filter list:

```
query = "SELECT * FROM books WHERE"
to_filter = []
```

Understanding the code

Then, if id, published, or author were provided as query parameters, we add ther
the query and the filter list:

```
    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
    if author:
        query += ' author=? AND'
        to_filter.append(author)
```

Understanding the code

If the user has provided none of these query parameters, we ha
nothing to show, so we send them to the “404 Not Found” page

```
if not (id or published or author):
    return page_not_found(404)
```

Understanding the code

To perfect our query, we remove the trailing ` AND and cap the query with required at the end of all SQL statements:

```
query = query[:-4] + ';'
```

Finally, we connect to our database as in our api_all function, then execute query we've built using our filter list:

```
conn = sqlite3.connect('books.db')
conn.row_factory = dict_factory
cur = conn.cursor()

results = cur.execute(query, to_filter).fetchall()
```

Understanding the code

Finally, we return the results of our executed SQL query as JSON
user:

return jsonify(results)

Understanding the code

Overall, the code:

1. Reads query parameters provided by the user
2. Builds an SQL query based on those parameters
3. Executes that query to find matching books in the database
4. Returns those matches as JSON to the user.

- This code makes our API's filtering capability more sophisticat
  users can now find books by, for example, Ursula K. Le Guin th
  published in 1975 or all books in the database published in 20

Pandas DataFrame as an API

- In the previous example, we published data from a database t
  API.
- It is not obvious how to publish data from a Pandas DataFram
  API using Flask.

Pandas DataFrame as an API

We will be looking at the Premier League data from the two seasons we lo
before. First, we import the modules needed, launch the app, and read in

```
import flask
from flask import request, jsonify
import pandas as pd


app = flask.Flask(__name__)
app.config["DEBUG"] = True


pl = pd.read_csv('C:/Users/DKITStaff/OneDrive - Dundalk Institute of
Technology/DKIT/Programming for Data
Analytics/Programming_2024_25/Datasets/pl_2seasons.csv')
```

Pandas DataFrame as an API

Then we set up a home route similarly to before:

```
@app.route('/', methods=['GET'])
def home():
        return '''<h1>Premier League Database</h1>
        <p>A prototype API for data on Premier League games.</
```

Pandas DataFrame as an API

Set up a route to show all data: /api/v1/leagues/pl/all. Define re
an empty dictionary that we fill in using a for loop. Finally, we js
the final result dictionary and return it.

```
@app.route('/api/v1/leagues/pl/all', methods=['GET'])
def api_all():
    result = {}
    for index, row in pl.iterrows():
        result[index] = dict(row)
    return jsonify(result)
```

Pandas DataFrame as an API

Create an error handler and run the app:

```
@app.errorhandler(404)
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not be found.</
404


app.run()
```

Exercise

- Allow filtering of the Premier League data frame through the A

- At the end of class I will show how to filter for certain columns equal to certain values eg HomeTeam = ‘Chelsea’ and AwayTea ‘Liverpool’, but if you want to include more advanced filtering project you will have to adapt the last example by yourself!