4 - BusBoard (Python)

Learning Goals

The goals of this exercise are to learn about:

- REST APIs, and how to call them
- Building a simple website

Problem background

Application Programming Interfaces (APIs) are everywhere. They provide a way for a developer to interact with a system in code. Look at the page here. It shows the next few buses due to stop near the Softwire office. This is great for an end user, but if you wanted to get this information and use it in your program, that page wouldn't be very useful.

Fortunately, there is a UK-wide transport API that provides an excellent set of endpoints which allow third-party developers (such as you) to get live data about many things, and use them in your applications. Your aim is to build an application which, given a postcode, finds the two nearest bus stops, and shows the next five buses due at each. We'd like to run this on a display near the office door so that people can see when the next buses are.

Getting started

First, register at the Transport API Portal. The easiest way to sign up is using GitHub, but you can also create a separate account (you may share an API key with others in your cohort for simplicity).

Now go to the API Reference page and view the documentation for API. This gives details of the different operations you can perform and lets you try them out.

For example, have a look at the `/uk/bus/stop/{atcocode}/live.json` endpoint.

- URI Parameters are request parameters that go in the path
- Query Parameters are those appended in the query string
- An example response. You should recognise it as JSON It contains a list of bus stops currently experiencing disruption. (There are probably quite a lot of them...)

You can try it out by taking the request URL and pasting it into your browser. You should see the same JSON response, probably not so nicely formatted (depending on your browser).

> You'll need to include your application id and key with every request, using the `app_id` and `app_key` parameters. These uniquely identify you, so make sure to be careful with them!

Having got that sorted, have a play about and see if you can work out how to get live arrival predictions for your nearest stop. (The stop code is 0180BAC30592). Do ask for help if you get stuck.

Part 1: Bus Times

Once you're returning sensible data for a given stop code, we need to turn this into a program. Your program should ask the user for a stop code, and print a list of the next five buses at that stop code, with their routes, destinations, and the time until they arrive in minutes.

We've built a framework project to get you started. It looks a bit empty now, but we'll bring in more exciting functionality later. Go to BusBoard on GitHub. **Click the "Fork" button** - this will create your own copy of the repository on GitHub. Now get a local copy of your repository:

```
cd \Work\Training
git clone git@github.com:YourNameHere/BusBoard-Python.git
```

Make sure you clone your own copy, not the CorndelWithSoftwire one, otherwise you won't be able to push any changes back to GitHub!

Open the project in your IDE as you did before. Now you should have a familiar-looking empty command-line project open.

Start by adding the necessary dependencies for calling a REST API. Python has built-in modules for making web requests, but there are libraries we can use that make things easier. We'll be using an external library called Requests to make the API requests, but using an in-built Python module to interpret the JSON response.

We will cover dependency management in more detail later, but for now it will suffice to know that Python comes with a utility called `pip` that you can use to download and install libraries for your application with ease. To do this, in your terminal, run the command "`pip install requests`".

Requests should now be installed and ready for use by your Python application. You will now want to learn how to use it. In general, when looking for Python libraries that contain the functionality you want, you should choose a library that is well documented (so that learning how to use them is as quick as possible), and popular (so you are more likely to get search results containing solutions to your problems). Accordingly, Requests is very popular, and its documentation is good too. A quickstart guide is available here.

Have a look at the methods available on the `requests` object and work out how to call the Transport API from your code. You will need to read the result, convert it into some sort of object, and then display the buses. The Python module you'll want to use to interpret the JSON response is the `json` module, and documentation is available here.

Make it so that your console application takes a bus stop code as an input, and prints out the next 5 buses at that stop.

Try to ensure you're using a sensible class structure with well-named methods. Remember to commit and push your changes as you go.

Part 2: Bus Stops

It would be really nice if we could just enter a postcode, rather than the stop code. Finding out stop codes is difficult, but everyone knows their postcode.

You will need to call two different APIs to make this happen:

1. http://api.postcodes.io allows you to find out lots of information about a given postcode.
2. The transport API lets you "get a list of Bus Stops near", which shows you bus stops near a given longitude and latitude.

See if you can work out how to wire these two APIs together. Then modify your program so that the user can supply a postcode and see the next buses at the two nearest bus stops.

Part 3: All Aboard

Now that you can convert a postcode into some buses, we want to display this in a simple website. Here's a website we made earlier:

```
git merge web-framework
```

You should find that you now have:

- Some files in the `static` and `templates` directories
- A new file `web.py`

First check you can still run your command-line app - it is very unlikely for there to have been a merge conflict so you should find everything works as expected.

Now try running the website - there are two ways you can start it up:

- If you were running your command-line app by typing `python main.py` into your terminal, you can run the website by typing `python web.py` there instead.
- Alternatively, If you are running the files directly from your IDE, then opening the `web.py` file and running that should do the same thing.

Once you've done either of those, go to http://localhost:5000 in your browser. You should find a simple website prompting you to enter a postcode - if not, ask a trainer for some help.

Sadly this website doesn't do much when you enter a postcode. We'll use the code you wrote for parts 1 and 2 to fix this.

Start by having a browse through the new code and files to see how it all works. The only parts you should need to change are the `busInfo()` method inside `web.py` and the `info.html` file. Don't worry about the `index.html`, `layout.html` or `bootstrap.min.css` files.

Now add code to the relevant places to get a working bus departure board. Look back to your `main.py` file to remind yourself how to join together the postcode and Transport APIs, and look in `web.py` for the right place to insert these so you can display them in `info.html`.

You may need to import classes or functions from other Python files in your project to do this. It's simple enough to do this: at the top of your file, write either `from filename import ClassName` (such as `from main import BusStop`) or `import filename` just as you may have done for external and built-in modules.

Need help with HTML or Jinja?

Code in the Templates folder is written in a templating language called Jinja. This is based on HTML - HyperText Markup Language, the language in which web pages are written - with some custom attributes added (the parts that are inside curly braces).

You only need very basic HTML skills to get this example working. If you've never come across it before, take a look at this W3Schools Introduction - just the first page should give you all the pieces you need to "fill in the blanks", although once you've got that working you may want to read on and learn more.

Jinja allows you to add attributes to html tags that are processed before the html is delivered to the user's browser. There are many possible instructions, but you'll only need some simple ones for this exercise. Let's take a look at how this works in the basic bus website.

In the `web.py` file, you'll see the code

```
return render_template('info.html', postcode=postcode)
```

This is saying that we want to return the file `info.html`, and also telling Jinja that it can access the object `postcode` with the name `postcode`.

Now look in `info.html` - you'll see the line

```
<p>You entered postcode <span>{{ postcode }}</span></p>
```

Most of that is regular HTML, but the `{{ postcode }}` is an instruction to Jinja to interpret the Python code within the braces and write the result of the expression `postcode`. In this case, we put a simple variable name, but any Python expression can be placed within the double curly braces, and its output will be written to the page.

One more Jinja delimiter you'll need to know about is this:

```
{% for item in items %}
 <div>Here is an item: {{ item }}</div>
{% endfor %}
```

A few things to note about this:

- We are using the `{% for x in y %}` Jinja construct to build a `for` loop for items we would like to repeat.
- The body of this `for` construct is a `<div>`, and it will be repeated on the page for each `item` in `items` - much like a normal `for` loop in Python!
- In Python, we normally do not need to end `for` loops since the interpreter is able to determine when the loop ends from the indentation in the file. This is not true for the Jinja template, so we need to explicitly end the loop using `{% endfor %}`.
- If `items` is an empty array, then the `<div>` will not be written to the page at all! Can you see why? Ask a trainer for an explanation if this does not make sense to you.

For more information on Jinja templating you can read their documentation here. But just by using `{{ expression }}`, `{% for x in y %}`, and some simple HTML, you should be able to display bus and arrival information in the website once you've entered a postcode.

Once you've got that working, look at the layout of your project - you have some code that is used by the website, some by the command-line app, and some by both - can you rearrange things into a set of files and classes so that these three concerns are separated?

Stretch goals

Here are some ideas for improvements - feel free to make more!

- Make your page refresh itself every 30 seconds.
- Display a nice error message if a user enters an invalid postcode.
- Improve the user interface design of the bus board.
- Publish your BusBoard to a free cloud hosting service.
- Think up some other neat things you could do with the Transport API...