

COM661 Full-Stack Strategies and Development

Practical B1: Introducing APIs and Flask

Aims

- To introduce the concept of an API as a software interface
- To demonstrate retrieval of data from a commercial API
- To introduce JSON data structures
- To manipulate data returned from an API in your own Python code
- To introduce Flask as a framework for development of Python web applications
- To examine the structure of a Flask application
- To develop API endpoints to return JSON data

Contents

B1.1 INTRODUCING APIS	2
B1.1.1 USING A COMMERCIAL API.....	3
B1.1.2 RETRIEVING CURRENT WEATHER DATA FROM THE OPENWEATHERMAP API.....	4
B1.1.3 RETRIEVING WEATHER FORECAST DATA	7
B1.2 INTRODUCING FLASK.....	10
B1.3 BUILDING A FIRST FLASK API	14
B1.3.1 RETRIEVING THE ENTIRE COLLECTION	15
B1.3.2 RETRIEVING A SINGLE ELEMENT	16
B1.4 FURTHER INFORMATION	19

B1.1 Introducing APIs

An Application Programming Interface (API) is a software library specification designed to be used as a means of enabling software components to communicate with each other.

Essentially, an API is an interface to some service or collection of data – except that rather than an interface designed to be used by a human operator, an API is an interface designed to be consumed by software.

In the Web environment, APIs are provided by many online platforms to allow content from an application or a service to be accessed over the HTTP protocol and embedded in 3rd party software. APIs have recently become a major growth area in web applications development, as seen by the *ProgrammableWeb* API Directory

(<https://www.programmableweb.com/category/all/apis>), which lists over 24,000 APIs in its collection. Many of these are freely available to use, some require a free user account to be created and others are made available on a subscription basis.

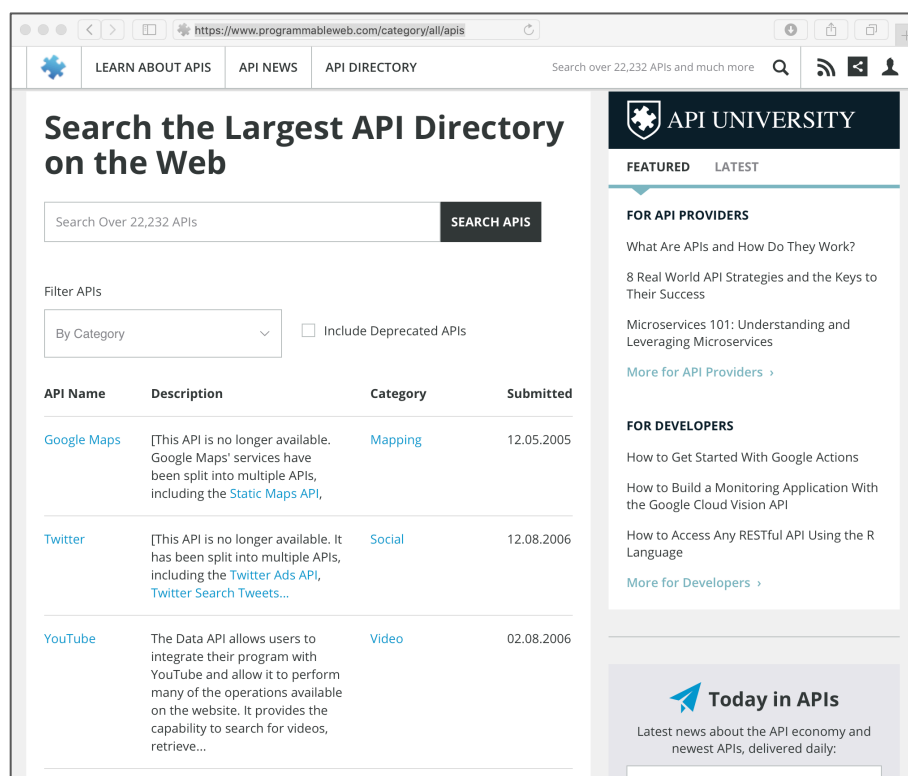


Figure B1.1 ProgrammableWeb API Directory

Most Web APIs are characterized by the term **REST (Representational State Transfer)**. This is a simple stateless architecture for the provision of web services that run over HTTP. Each client request to a REST service contains all information required to service the request and no client context information is maintained by the service provider. REST services are

defined by an **endpoint** (URL), a **media type** to be returned (XML, JSON, etc.) and the **HTTP method** to be used (GET, POST, etc.). REST services are consumed by making an HTTP request to the endpoint, and accepting the data returned. We will examine the structure and use of REST (sometimes referred to as **RESTful**) APIs over the course of this section of the module.

In this practical, we will first examine the facilities provided by a typical commercial API and see how we can retrieve real-time data for use in our own applications. However, our focus will be on designing and building our own APIs, so we will also introduce **Flask** – a Python package that makes it easy to provide an API that can manage access to a data source of our own.

B1.1.1 Using a Commercial API

The **OpenWeatherMap** REST API is a collection of URLs (known as **endpoints** in API terminology) that enable us to write code to search an online weather reporting resource and manipulate the results for presentation in our own applications. The API provides very powerful retrieval functionality and allows us to manipulate the weather data that is returned within our own code – but is only available to registered developers who have signed up using the following process.

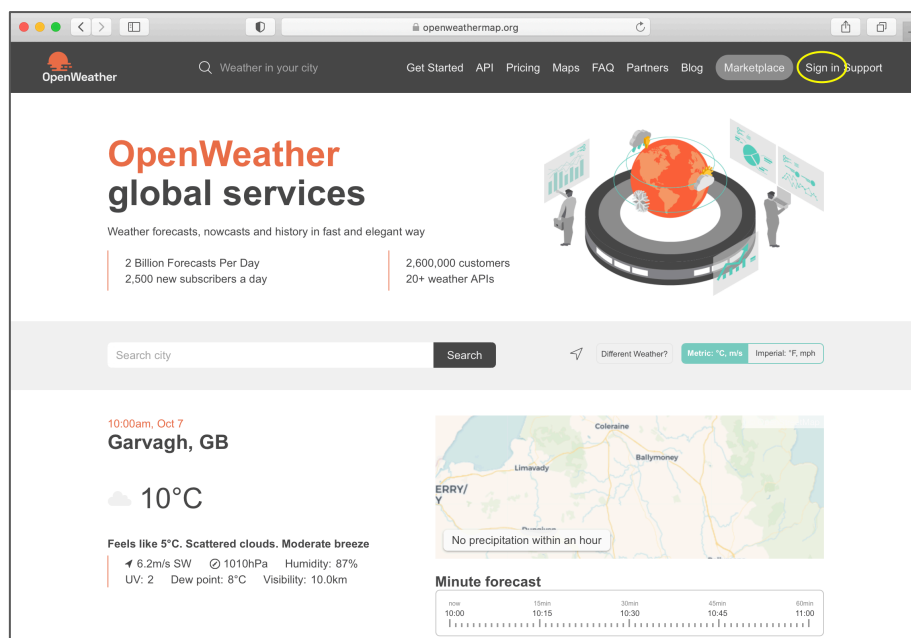


Figure B1.2 OpenWeatherMap

1. Go to <https://openweathermap.org/> and click the “Sign In” icon at the top of the browser display (highlighted in Figure B1.2 above)
2. Complete the application process until you are logged in and see the main menu near the top of the window as illustrated in Figure B1.3.

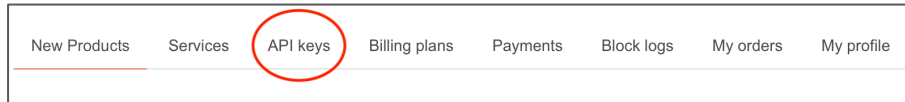


Figure B1.3 OpenWeatherMap Main Menu

3. Click on the **API Keys** option in the main menu (highlighted in Figure B1.3 above) and take a note of your API key. You will need this when making calls to the API from your applications.

Do it now!	Follow the steps above to sign up to OpenWeatherMap and generate your Access Key.
-------------------	--

B1.1.2 Retrieving Current Weather Data from the OpenWeatherMap API

As a first example, we will create a simple application that calls the `/data/2.5/weather` endpoint to return the current weather conditions. The sample program `currentWeather.py` provides a first demonstration of an application that queries the **OpenWeatherMap** API to harvest the most recently reported weather conditions at the location specified by the latitude and longitude values that correspond to Coleraine.

The key functions here are `url_builder()` and `fetch_data()` that respectively compose the URL to be accessed and retrieve the JSON data from that location. The function `url_builder()` takes the stem of the API endpoint (<http://api.openweathermap.org/data/2.5/weather>) and appends parameters as a querystring as follows:

Parameter	Values
mode	Determines the format of the data to be returned. Options are json (the default), xml or html
unit	imperial for temperature in Fahrenheit, metric for temperature in Celsius, no value (default) for temperature in Kelvin
APPID	the user’s API key

lat	The latitude value of the forecast location
lon	The longitude value of the forecast location

File: B1/current_weather.py

```
import datetime
import json
import urllib.request

def url_builder(lat, lon):
    user_api = 'add_your_api_key_here'
    unit = 'metric'
    return 'http://api.openweathermap.org/' + \
        'data/2.5/weather'+ \
        '?units=' + unit + \
        '&APPID=' + user_api + \
        '&lat=' + str(lat) + \
        '&lon=' + str(lon)

def fetch_data(full_api_url):
    url = urllib.request.urlopen(full_api_url)
    output = url.read().decode('utf-8')
    return json.loads(output)

def time_converter(timestamp):
    return datetime.datetime.fromtimestamp(timestamp). \
        strftime('%d %b %I:%M %p')

lat = 55.147872
lon = -6.675298
json_data = fetch_data(url_builder(lat, lon))

temperature = str(json_data['main']['temp']);
timestamp = time_converter(json_data['dt'])
description = json_data['weather'][0]['description']

print("Current weather")
print(timestamp + " : " + temperature + " : " + \
    description)
```

Note: As well as latitude and longitude, we can also search **OpenWeatherMap** by place name, country, zip-code, city ID or within a region specified as a bounding box. See the API endpoint documentation at <https://openweathermap.org/current> for full details

The endpoint URL produced by `url_builder()` is passed to `fetch_data()` which opens the url and retrieves the data as a string. The string is then deserialized to a Python object by the `loads()` method provided by Python's `json` package.

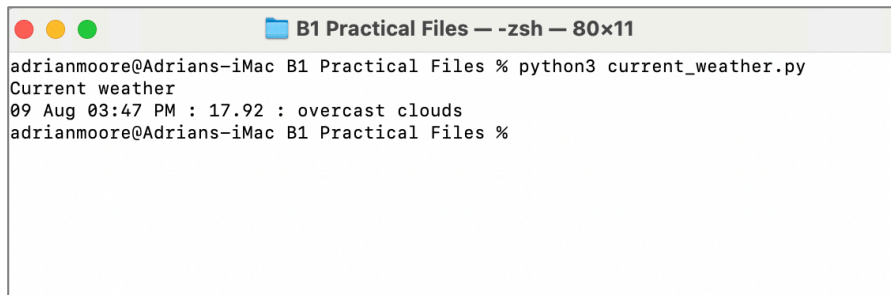
The data returned from the API call is a complex JSON object that contains many fields. You can see the full list in the API documentation, but some of the main elements are described below.

```
{
  "name"      : "name of location",
  "dt"        : "time of data calculation as a Unix UTC value"
  "weather"   : [
    {
      "main"      : "headline description",
      "description" : "more detailed description",
      "icon"       : "weather symbol icon ID"
    }
  ],
  "main"      : {
    "temp"        : "temperature reading",
    "pressure"     : "air pressure reading",
    "humidity"     : "air humidity reading"
  },
  "sys"       : {
    "sunrise"      : "sunrise time as a Unix UTC value",
    "sunset"       : "sunset time as a Unix UTC value"
  }
}
```

Note:	JSON (JavaScript Object Notation) is an open standard that uses human-readable text to transmit data consisting of attribute/value pairs. It is a very convenient structure for Python developers as the format of JSON maps very closely to the structure of a Python dictionary. It is most often used in web applications as an alternative to XML. Further information on JSON can be found in Section B1.5
--------------	---

Finally, we retrieve the information to be displayed from the data structure, converting the UTC timestamp to a more readable form by the `time_converter()` function. You can find more information on the codes used in the Python `strftime()` function to convert Unix UTC values to readable form in Section B1.5.

Do it now! Add your API key to the code (line 6). Now, run *current_weather.py* and verify that it displays the current weather information as illustrated in Figure B1.4 below.

A terminal window titled "B1 Practical Files — -zsh — 80x11" showing the execution of a Python script. The prompt is "adrianmoore@Adrians-iMac B1 Practical Files %". The command entered is "python3 current_weather.py". The output is "Current weather" followed by a new line and "09 Aug 03:47 PM : 17.92 : overcast clouds". The prompt returns to "adrianmoore@Adrians-iMac B1 Practical Files %".

```
adrianmoore@Adrians-iMac B1 Practical Files % python3 current_weather.py
Current weather
09 Aug 03:47 PM : 17.92 : overcast clouds
adrianmoore@Adrians-iMac B1 Practical Files %
```

Figure B1.4 Using the Current Weather API Endpoint

Try it now! Modify the code so that (i) the name of the location is retrieved from the data structure and added to the title message so that it reads “Current weather in Coleraine”; and (ii) the forecasted sunset time is displayed

Try it now! Write the Python program *compare_temperatures.py* that prompts the user for two latitude and longitude values representing a pair of locations and uses the **OpenWeatherMap** API to retrieve the location name and current temperature for each. The program should output the result in the form (e.g.) “Coleraine is hotter than Berlin”. You can obtain latitude and longitude values for any location from Google Maps or by searching for the location on Wikipedia.

B1.1.3 Retrieving Weather Forecast Data

The sample program *forecast.py* illustrates the API endpoint that retrieves weather forecast information for a specified location. This time, the API returns a data structure that contains 3-hourly forecasts for a 5-day period (i.e. up to 40 forecasts in total)

File: B1/forecast.py

```
...

def url_builder(lat, lon):

    ...
    return 'http://api.openweathermap.org/' + \
        'data/2.5/forecast'+ \
        '?mode=' + mode + \
        '&units=' + unit + \
        '&APPID=' + user_api + \
        '&lat=' + str(lat) + \
        '&lon=' + str(lon)

def fetch_data(full_api_url):
    ...

def time_converter(timestamp):
    ...

...

print("Weather forecast for " + \
      json_data['city']['name'])
print(str(json_data['cnt']) + " forecast(s) retrieved")
for forecast in json_data['list']:
    timestamp = time_converter(forecast['dt'])
    temperature = str(forecast['main']['temp'])
    description = forecast['weather'][0]['description']
    print(timestamp + " : " + temperature + \
          " : " + description)
```

This time, the code that generates the output is in a loop that iterates across of the members of the list structure, retrieving a new forecast in each iteration. The fields available in the returned data are similar to the **weather** endpoint – except that the location information is presented only once, and the individual forecasts are in a list.

The complete specification of the data set returned can be seen in the online API documentation, but all of the main elements (including all those used in the code presented here) are shown in the following JSON structure.


```

{
  "city": {
    "id" : "location ID",
    "name" : "name of location",
    "coord" : {
      "lon" : "longitude value",
      "lat" : "latitude value"
    }
  }
  "cnt" : "number of forecasts returned"
  "list" : [
    {
      "dt" : "time of data calculation as a Unix UTC value"
      "weather": [
        {
          "main": "headline description",
          "description": "more detail",
          "icon": "weather symbol icon ID"
        }
      ],
      "main" : {
        "temp" : "temperature reading",
        "pressure" : "air pressure reading",
        "humidity" : "air humidity reading"
      },
    }
  ]
}

```

Do it now!	Add your API key to the source code (line 6). Now, run <i>forecast.py</i> and verify that it displays the forecast information for Coleraine as illustrated in Figure B1.5 below.
-------------------	---

```

B1 Practical Files — -zsh — 80x24
adrianmoore@Adrians-iMac B1 Practical Files % python3 forecast.py
Weather forecast for Coleraine
40 forecast(s) retrieved
09 Aug 04:00 PM : 17.92 : light rain
09 Aug 07:00 PM : 17.12 : light rain
09 Aug 10:00 PM : 15.19 : overcast clouds
10 Aug 01:00 AM : 13.02 : broken clouds
10 Aug 04:00 AM : 12.78 : scattered clouds
10 Aug 07:00 AM : 13.21 : broken clouds
10 Aug 10:00 AM : 14.09 : overcast clouds
10 Aug 01:00 PM : 15.6 : overcast clouds
10 Aug 04:00 PM : 16.72 : scattered clouds
10 Aug 07:00 PM : 16 : light rain
10 Aug 10:00 PM : 13.96 : overcast clouds
11 Aug 01:00 AM : 14.44 : overcast clouds
11 Aug 04:00 AM : 14.52 : overcast clouds
11 Aug 07:00 AM : 14.68 : overcast clouds
11 Aug 10:00 AM : 14.34 : light rain
11 Aug 01:00 PM : 16.05 : moderate rain
11 Aug 04:00 PM : 17.19 : light rain
11 Aug 07:00 PM : 15.78 : broken clouds
11 Aug 10:00 PM : 12.76 : clear sky
12 Aug 01:00 AM : 11.7 : clear sky
12 Aug 04:00 AM : 11.76 : scattered clouds

```

Figure B1.5 Using the Forecast API Endpoint

Try it now!	Modify the code so that it only presents the highest temperature forecast on each day. If more than one forecast for a day shares the highest temperature, then the first time that temperature is achieved should be shown.
--------------------	--

B1.2 Introducing Flask

Flask is a framework for the development of web applications written in Python. It is classified as a micro-framework as it does not require any particular tools or libraries but makes it very easy to quickly create working useful applications. A Web Application Framework is *“a software architecture designed to support the development of dynamic websites, web applications and web services”* (Ref:

http://en.wikipedia.org/wiki/Web_application_framework). In effect, it is a library that hides the basic functions of the underlying gateway, by providing a range of functions or class libraries that promote rapid development and code reuse.

Using any external library in an application always raises a potential issue if that library is updated during the lifetime of our program. It might be the case that an update to a library creates a breaking change that prevents our program from running, in which case we would refuse the update – but we might also want to install the update in order to take advantage of new features in future applications that we will develop.

The answer in Python is to use a **virtual environment** to contain each application that uses external libraries. By housing each application within its own container, each can have the versions of the external libraries that are applicable to it – without compromising the run-time environment of any other applications.

This is such a common situation with Python that the language is packaged with its own virtual environment module called **venv**. Creating a new Flask application is therefore a 4-stage process as illustrated below by the creation of a new application called “first”:

Note:	On the lab machines, Virtual Studio Code may default to a PowerShell terminal (identified by “PS” at the beginning of the prompt). You should make sure that you are using a “basic” cmd shell for Virtual Environment work.
--------------	--

1. **Create the folder** in which your application will reside and navigate into it

Windows	MacOS
<code>mkdir first</code> <code>cd first</code>	<code>mkdir first</code> <code>cd first</code>

2. **Create a virtual environment** and launch it

Windows	MacOS
<code>python -m venv venv</code>	<code>python -m venv venv</code>

In this command we are using the python interpreter to make (-m) a new **venv** instance called **venv**. You are not required to call the environment **venv**, but it is the most common approach.

3. **Activate** the virtual environment

Windows	MacOS
<code>venv\Scripts\activate</code>	<code>. venv/bin/activate</code>

Once the virtual environment has been launched, you should see its name (**venv**) at the beginning of the command prompt.

4. **Install the Flask module.** This is most easily done using the pip Package Manager that comes bundled with Python

Windows	MacOS
<code>pip install flask</code>	<code>pip install flask</code>

Do it now!	Follow steps 1-4 above to create the environment for our first Flask application.
-------------------	---

To prove that everything is correctly installed, we can create a basic “Hello World” application. Create a new file within the *first* folder called *app.py* as follows.

Note: The version of *app.py* provided in the Practical Files is that at the end of this practical. You should begin with a blank file at this point and build up the code as you work through this material. You are encouraged to enter the code at the keyboard rather than use copy and paste.

File: B1/first/app.py

```
from flask import Flask, make_response

app = Flask(__name__)

@app.route("/", methods=["GET"])
def index():
    return make_response("<h1>Hello world</h1>", 200)

if __name__ == "__main__":
    app.run(debug=True)
```

There are four distinct phases to this code as follows

1.

```
from flask import Flask, make_response
```

Import the elements of the Flask package that will be needed in this application. We will see how these are used in the later stages.

2.

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

Create the Flask application. This line of code simply creates the application object as an instance of class **Flask** imported from the **flask** package. The **__name__** variable passed to the class is a Python predefined variable, which is set to the name of the module in which it is used.

3.

```
@app.route("/", methods=["GET"])
def index():
    return make_response("<h1>Hello world</h1>", 200)
```

This is effectively the main program body, where we specify an **@app.route** decorator and a function. We will look more closely at Python decorators later, but here we are stating that if a GET request is received to the root address of the website, then the function provided should be run. Note that the name of the function is completely arbitrary – it is not called by name but is run automatically when a request to the specified URL is received.

The body of the function returns a value by the Flask **make_response()** function. Here, this takes two parameters – the HTML to be returned in response to the request and the HTTP status code returned to the browser. Here we return code 200 denoting a successful call – but we will see other codes and when to use them later.

4.

```
if __name__ == "__main__":
    app.run(debug=True)
```

Finally, we have the standard Python statement that runs the application. The role of the **if** statement here is to make sure that this application can only be invoked directly (e.g. from the command line). This is useful when your application contains multiple source files, but your code has a single entry point that should be used.

Do it now!

Follow steps 1-4 above to code and run the first Flask application. When the application is running, visit <http://localhost:5000/> in a web browser and ensure that you obtain output such as that shown in Figure B1.6.

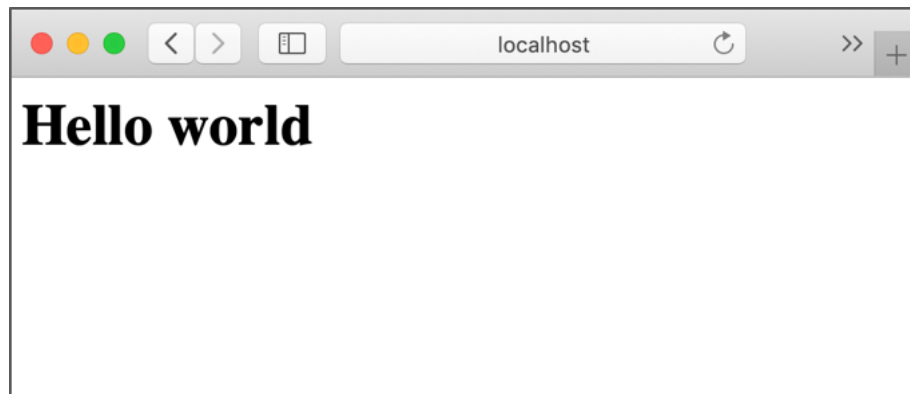


Figure B1.6 “Hello world” Flask application

B1.3 Building a First Flask API

Now that our first Flask application is running, we will modify it to act as an API that provides access to a collection of data about businesses. Each business is described by a unique **id** value, a **name**, a **town**, a **star** rating out of 5 and a collection of **reviews** (initially empty). In time, we will see how to manipulate this data from a live database, but initially we will hard-code that data as a JSON object stored in a list of Python dictionaries.

The initial state of our API is presented in the code box below. Note that we have added the **jsonify** function to the list of elements to be imported from the Flask module (line 1) and we have also removed the **@app.route()** that supports the “Hello World” functionality.

File: B1/first/app.py

```
from flask import Flask, jsonify, make_response

app = Flask(__name__)

businesses = [
    {
        "id" : 1,
        "name" : "Pizza Mountain",
        "town" : "Coleraine",
        "rating" : 5,
        "reviews" : []
    },
    {
        "id" : 2,
        "name" : "Wine Lake",
        "town" : "Ballymoney",
        "rating" : 3,
        "reviews" : []
    },
    {
        "id" : 3,
        "name" : "Sweet Desert",
        "town" : "Ballymena",
        "rating" : 4,
        "reviews" : []
    }
]

if __name__ == "__main__":
    app.run(debug=True)
```

Do it now!	Modify <i>app.py</i> as shown above. You can replace the values in the businesses list with any data of your own and add additional business entries to increase the size of the data set.
-------------------	---

B1.3.1 Retrieving the entire collection

Now, we add the route and function that provide the initial functionality that retrieves details of all businesses in the collection. As you will have seen in the **OpenWeatherMap** API, it is good practice to include the API name and version number in the URL (so that your host can support multiple versions of multiple APIs if required), so we adopt that practice here by specifying that the URL to retrieve the collection of businesses is **/api/v1.0/businesses**.

The function to return all businesses is very straightforward and simply requires us to return the entire **businesses** object, passing it through the Flask **jsonify()** function so that it is returned in JSON form.

File: B1/first/app.py

```
...

@app.route("/api/v1.0/businesses", methods=["GET"])
def show_all_businesses():
    return make_response( jsonify( businesses ), 200 )

if __name__ == "__main__":
    app.run(debug=True)
```

Do it now!

Add the code shown above to *app.py* and run the application (the server should automatically re-load when you change the source code, but if not, you can use CTRL-C to quit the previous server if it is still running). Now visit the URL <http://localhost:5000/api/v1.0/businesses> and make sure that you are presented with the entire businesses collection as shown in Figure B1.7 below.

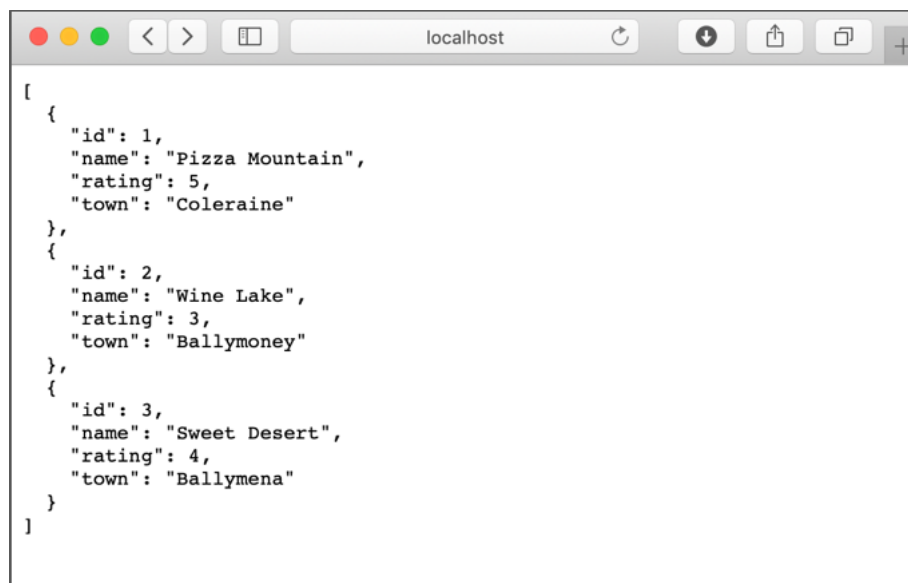


Figure B1.7 Retrieving all businesses

B1.3.2 Retrieving a single element

Next, we will add the ability to retrieve a single element from the collection by specifying it by **id** in the URL. Therefore, to request the business with **id** value 1, the URL is **/api/v1.0/businesses/1**, and so on.

This requires that we add a new `@app.route()` decorator and corresponding function to the application as provided below.

File: first/app.py

```
...

@app.route("/api/v1.0/businesses/<int:id>", methods=[ "GET" ])
def show_one_business(id):
    data_to_return = \
        [ business for business in businesses \
          if business["id"] == id ]
    return make_response( jsonify( \
                          data_to_return[0] ), 200 )

if __name__ == "__main__":
    app.run(debug=True)
```

The first thing to notice in this new code is the specification of a variable parameter in the URL. Here, we know that the URL will contain an integer `id` value, but we don't know exactly what the value will be, so we specify it in the URL as `<int:id>`. The variable element (`id` in this case) will then be specified as a parameter to the function that is run in response to a request to that URL.

There are many ways in which we could extract the element to be returned, but here we show a construction that is very typical of Python. Essentially, the task is to generate a list of all businesses where the `id` value matches that provided as a parameter, so we build up the expression as follows

1. We want to return a list

```
[      ]
```

2. We want to return a list of elements called **business**

```
[ business ]
```

3. We want to return a list of elements where each element is a member of the **businesses** collection

```
[ business for business in businesses ]
```

4. We are only interested in the elements where the **id** value matches that passed as the parameter to the function

```
[ business for business in businesses \
    if business["id"] == id
]
```

Finally, we only return element **[0]** of the list retrieved, since only one business should match the **id** value in the URL.

Do it now! Add the code shown above to `app.py` and run the application. Now visit the URL <http://localhost:5000/api/v1.0/businesses/1> and make sure that you are presented with the single business requested as shown in Figure B1.8 below.

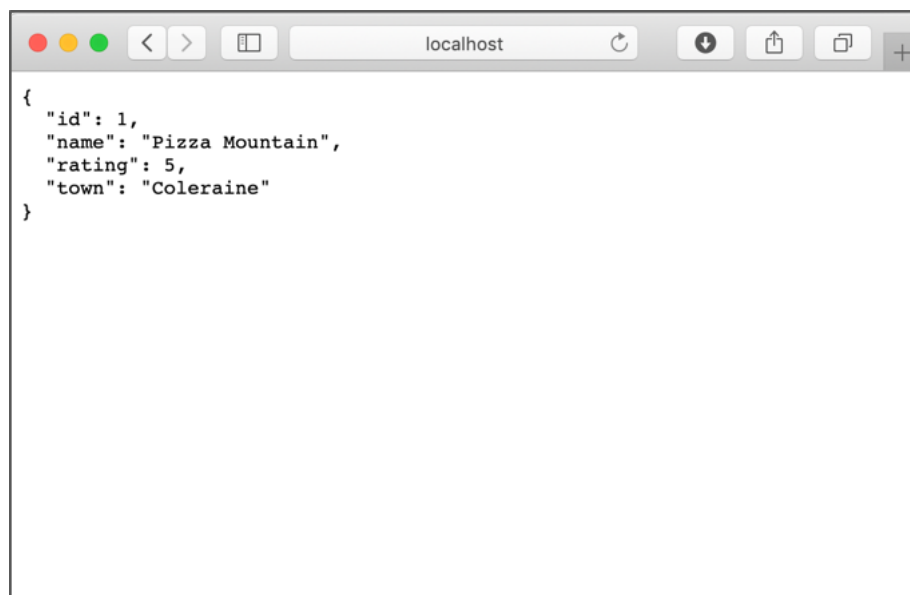


Figure B1.8 Retrieve details of a single business

B1.4 Further Information

- <https://www.programmableweb.com/category/all/apis>
The ProgrammableWeb API Directory
- <https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f>
What exactly IS an API?
- <https://techterms.com/definition/api>
API definition from TechTerms
- <http://vrisci.lojban.org/~cowan/restws.pdf>
RESTful Web Services
- <https://flask.palletsprojects.com/en/2.0.x/>
Flask – the documentation
- <https://pythonspot.com/flask-hello-world/>
Getting started with Flask – Hello World
- <https://openweathermap.org>
OpenWeatherMap home page
- <http://www.infoq.com/articles/rest-introduction>
A brief introduction to REST
- <http://code.tutsplus.com/tutorials/a-beginners-guide-to-http-and-rest--net-16340>
A beginner's Guide to HTTP and REST
- <http://www.youtube.com/watch?v=7YcW25PHnAA>
REST API: Concepts and Examples (YouTube)
- <http://www.json.org>
Specification of the JSON notation
- http://www.w3schools.com/js/js_json_intro.asp
W3Schools JSON tutorial
- <http://strftime.org>
Codes for converting Unix RTC time to readable form