COM661 Full Stack Strategies and Development

BE04. Improving the Basic API

Aims

- To recognise the limitations of the basic API produced previously
- To introduce the Python `uuid` (**U**niversal **U**nique **ID**entifier) library
- To demonstrate the generation of a randomised data set
- To appreciate the performance improvements associated with dictionaries
- To demonstrate the detection of errors in API calls and the generation of appropriate responses
- To implement pagination in the return of a large data set

Table of Contents

4.1 More Flexible Data Retrieval

In the previous practical, we built a fully functional API that supported CRUD operations on data relating to a set of businesses, where each business object also includes a nested sub-document consisting of a set of reviews contributed by users. We also saw how Postman provides a very useful tool for testing API requests such as POST, DELETE and PUT without having to build an HTML front-end.

However, the API is quite basic in some ways and before proceeding to replace the hard-coded data set with a live database, we will identify the major weaknesses on our provision and identify ways in which we can address them.

4.1.1 Generating Unique Key Values

Our basic provision used a simple incremental count to assign unique id values to businesses and reviews. This is a useful starting point, but is not a good long-term solution for two reasons:

i) Without implementing an **`id`** management scheme, it is possible for previously used **`id`** values to be assigned to new businesses. For example, say we have 10 businesses with **`id`**s numbered from 1 to 10. If we then delete the business with **`id`** value 10 and then issue a POST request to create a new business, the value 10 will be assigned to that new business. This is not necessarily a problem, but it is better practice that an **`id`** is assigned to an object for life – and if that object should be deleted, the **`id`** value is never re-assigned.

ii) More seriously, our update and delete operations require we loop across all of the businesses (and reviews) in order to find the one to which we want to apply the operation. For a small collection of businesses, this might be acceptable, but if the collection grows to 100s (or even more), then this will be a significant overhead.

We will therefore make two changes to our object structure – we can use the Python **`uuid`** (**U**niversal **U**nique **ID**entifier) package to generate a unique value to be used as the **`id`** and we will structure the collection of businesses as a dictionary rather than a list, to give direct access to a business if the **`id`** value is known.

The Python method **`uuid.uuid1()`** generates a 128-bit unique value as a combination of the MAC address of the host computer and the current time. The combination of MAC

address and time guarantees that values will be unique across different applications and different machines.

As a first step, we import **`uuid`** into our application and use it to generate the **`id`** values in our data set. Note that we convert the 128-bit value to a string so that we can use it as part of a URL, so we also need to change all of the instances of business ID within **`@app.route()`** decorators to denote that the variable parameter is now a **`string`** rather than an **`int`**.

File: app.py

```
from flask import Flask, jsonify, make_response, request
import uuid

...

businesses =  [
    {
        "id" : str(uuid.uuid1()),
        "name" : "Pizza Mountain",
        "town" : "Coleraine",
        "rating" : 5,
        "reviews" : []
    },

    ...
]

...

@app.route("/api/v1.0/businesses/<string:id>", \
           methods=["GET"])
def show_one_business(id):

...
```

Do it now!	Make the modifications shown above to incorporate **`uuid`** values as identifiers. Be sure to change all of the business ID parameters in all **`@app.route()`** decorators. Use Postman to ensure that the GET, POST, PUT and DELETE functionality on businesses still works as previously.

Try it now!	Make the same modification to the **review ID** values. Once again, use Postman to make sure that all functionality still works as previously.

4.1.2 A More Realistic Dataset

Our hard-coded data set was sufficient to test the basic operation of the API, but for a more realistic simulation we require more data. At the same time, we will re-structure the collection of businesses so that it is a dictionary rather than a list with the **`uuid`** value used as the key field for each business. The function **`generate_dummy_data()`** generates a dictionary of 100 business objects, where each object has a key field of a **`uuid`** string and a value of a dictionary of *name*, *town*, *rating* and *reviews*

File: app.py

```
from flask import Flask, jsonify, make_response, request
import uuid, random

...

businesses = {}

def generate_dummy_data():
    towns = ['Coleraine', 'Banbridge', 'Belfast',
             'Lisburn', 'Ballymena', 'Derry', 'Newry',
             'Enniskillen', 'Omagh', 'Ballymoney']
    business_dict = {}

    for i in range(100):
        id = str(uuid.uuid1())
        name = "Biz " + str(i)
        town = towns[random.randint(0, len(towns)-1)]
        rating = random.randint(1, 5)
        business_dict[id] = {
           "name" : name,
           "town" : town,
           "rating" : rating,
           "reviews" : []
        }
    return business_dict

...

if __name__ == "__main__":
    businesses = generate_dummy_data()
```

To randomize the data set, the *town* is selected at random from a list of 10 locations and the *rating* is set to a random value in the range 1-5. The *name* is a concatenation of the string "Biz " and the loop counter so that each of the 100 businesses has a different name. To generate the data, we call the new function from the main section at the bottom of the code.

Do it now!	Add the new code shown above to the application and verify its effect by loading the URL http://localhost:5000/api/v1.0/businesses to a web browser to see the complete list of business objects. Verify that you receive output similar to that shown in Figure 4.1 below.

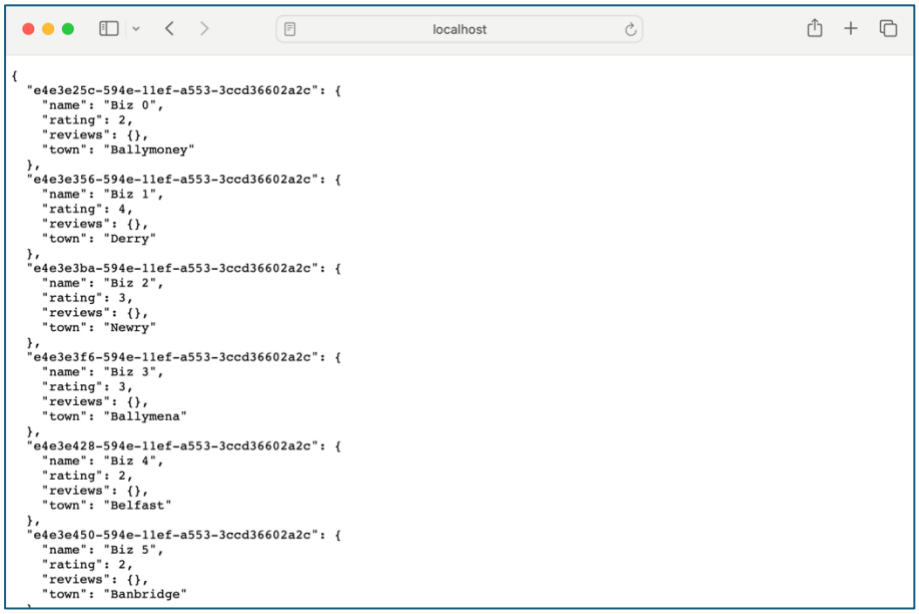

Figure 4.1 Random Dataset Organised as a Dictionary

Note how the data set above is now a dictionary, where the key field of each entry is the random **`uuid`** value. This allows us to significantly simplify the code in the functions that support each route – while also benefitting from an even more significant performance improvement.

4.1.3 Updating the Application Logic

Previously, when returning a single business, we had to loop through the collection to find the entry where the **`id`** value matched that passed as a parameter. With a dictionary, access it much more efficient since we can simply directly access the required dictionary element by using the key value as the index.

File: app.py

```
...

def show_one_business(id):
    return make_response( jsonify(  businesses[id] ), \
                          200 )


...
```

Do it now!	Make the modification to the **`show_one_business()`** function shown above and check that the function still works as expected.

The code to add a new business only has minor changes from the previous version. Rather than calculate the new **`id`** value to be used, we generate it with a call to **`uuid.uuid1()`** and create a new dictionary entry with the new **`id`** value as the key. Finally, we return a JSON object consisting of the new **`id`** as the key and the body of the new business object as the value.

File: app.py

```
...

def add_business():
    next_id = str(uuid.uuid1())
    new_business = { "name" : request.form["name"],
                     "town" : request.form["town"],
                     "rating" : request.form["rating"],
                     "reviews" : []
                    }
    businesses[next_id] = new_business
    return make_response( \
           jsonify( { next_id : new_business } ), 201


...
```

Do it now!	Make the modification to the **`add_business()`** function shown above and check using Postman that the function still works as expected.

Editing a business is another example of the performance efficiency provided by using dictionaries. Previously we had to loop across the business objects comparing the **`id`** of

each, while we can now directly access the element, changing the values to those provided by the user.

File: app.py

```
...

def edit_business(id):
    businesses[id]["name"] = request.form["name"]
    businesses[id]["town"] = request.form["town"]
    businesses[id]["rating"] = request.form["rating"]
    return make_response( \
           jsonify( { id : businesses[id] } ), 200 )
...
```

Do it now!	Make the modification to the **`edit_business()`** function shown above and check using Postman that the function still works as expected.

Note:	Remember that each time the application is run, the set of businesses will be generated anew. This means that **`id`** values from a previous run will no longer be valid, so you will need to display all businesses to get an **`id`** to use when testing the edit function.

Finally, the code to delete a business is reduced to two simple statements – one to remove the dictionary entry with the specified key (**`id`** value) and another to send the empty JSON response.

File: app.py

```
...

def delete_business(id):
    del businesses[id]
    return make_response( jsonify( {} ), 204)

...
```

Do it now!	Make the modification to the **`delete_business()`** function shown above and check using Postman that the function still works as expected.

Try it now!	Update the functions to retrieve, add, edit and delete **reviews** in the same way. Make sure that reviews are structured as a dictionary with each review identified by a **`uuid`** value in the same way as businesses. **Hint:** Remember that you will need to change the definition of the reviews element in **`generate_dummy_data()`** and **`add_business()`** to a dictionary rather than a list.

4.2 Validation and Error Handling

So far, we have assumed that all values provided to API calls are correct. For example, see what happens when you provide an invalid business ID to http://localhost:5000/api/v1.0/businesses/id

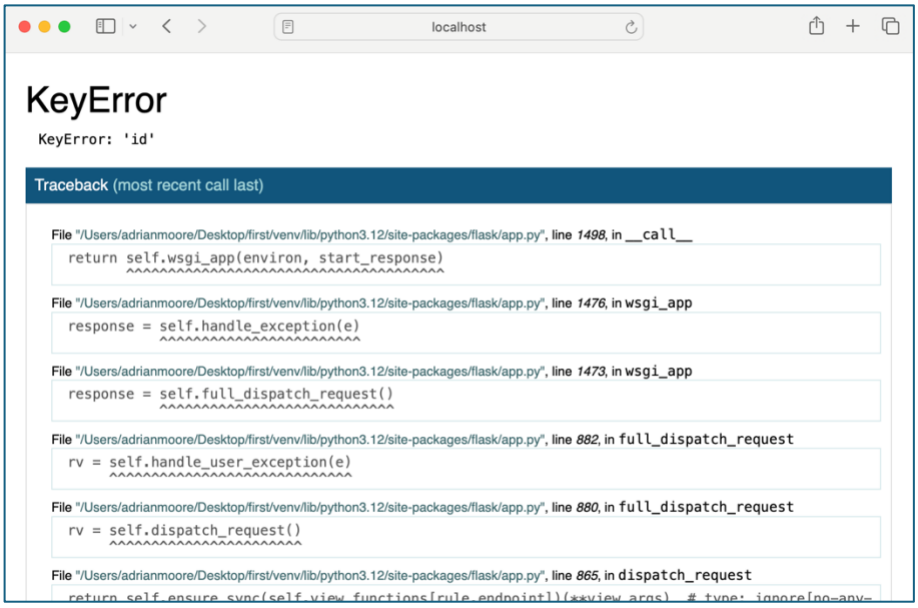

Figure 4.2 Invalid ID Value in URL

What we see in the browser in Figure 4.2 is the standard Flask/Python error message generated when we attempt to look up a dictionary using a key value that does not match any of the dictionary entries. We want to improve our API by generating a more meaningful response when an invalid operation is attempted.

As a first step, we will trap the *bad id* error in the GET request for an individual business. Before generating the response, we first test to see if the specified key value is present in the dictionary. If it exists, then we can generate the response as before, but if not, we

generate a JSON error message that can be returned along with the **404** status code that denotes *page not found*.

File: app.py

```
...

def show_one_business(id):
    if id in businesses:
        return make_response( jsonify(  \
                                businesses[id] ), 200 )
    else:
        return make_response( jsonify( \
          { "error" : "Invalid business ID" } ), 404 )

...
```

Do it now!	Make the modification to the **`show_one_business()`** function shown above and check using Postman that the function still works as expected.

With the application running, repeat the previous experiment to retrieve a business with an invalid **`id`** value. Check that you can see the error message and status code returned as illustrated in Figure 4.3 below.

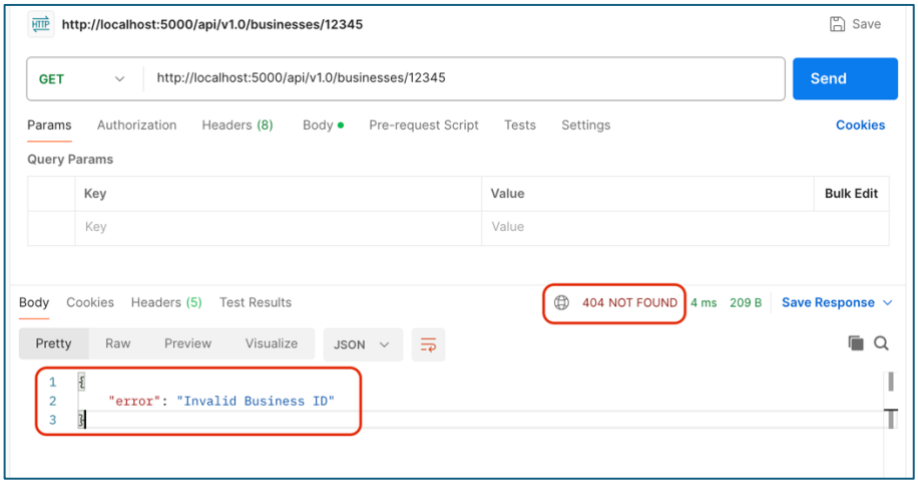

Figure 4.3 Error Handling

When adding a new business, the main potential for error is when the user does not provide all of the information that is required. Obviously, in a real application some fields may be optional, but in our example, we will assume that all of *name*, *town* and *rating* are needed before the business can be added.

In order to implement this, we check that the required values are present in the **`request.form`** object, only generating the new business if this is the case. Where any of the parameters are missing, we generate an error message.

File: app.py

```
...

def add_business():
    if "name" in request.form and
       "town" in request.form and
       "rating" in request.form:
        next_id = str(uuid.uuid1())
        new_business = {
            "name" : request.form["name"],
            "town" : request.form["town"],
            "rating" : request.form["rating"],
            "reviews" : {}
        }
        businesses[next_id] = new_business
        return make_response( jsonify(
               { next_id : new_business } ), 201 )
    else:
        return make_response( jsonify(
               { "error" : "Missing form data" } ), 404 )

...
```

Do it now!	Test the new functionality by adding a new business with all values provided and with some missing to ensure that the error trapping is working properly.

Note:	A further level of error trapping might be to test the type and range of each parameter to check that (for example) the rating is an integer between 1 and 5. This level of checking is left for you to complete as an exercise.

When editing an existing business entry, there are two different errors that we want to trap (i) when the **`id`** provided does not match a business in the collection and (ii) when all of the required values for the edit operation are not provided. This makes for a slightly more complex scenario than the previous examples, where a pair of nested **`if`** statements are used to check for the potential errors.

File: app.py

```
...

def edit_business(id):
    if id not in businesses:
        return make_response( jsonify(
           { "error" : "Invalid business ID" } ), 404 )
    else:
        if "name" in request.form and
           "town" in request.form and
           "rating" in request.form:
            businesses[id]["name"] = request.form["name"]
            businesses[id]["town"] = request.form["town"]
            businesses[id]["rating"] =
                                    request.form["rating"]
            return make_response( jsonify(
                   { id : businesses[id] } ), 200 )
        else:
            return make_response( jsonify(
               { "error" : "Missing form data" } ), 404 )

...
```

Do it now!	Make the changes shown above and test the new functionality by providing input such that all three potential paths are followed.

Try it now!	Update **`edit_business()`** so that only the fields provided by the user are modified. For example, if a PUT request were received and only the rating value was provided, then the operation would be successful with the rating updated to the new value and the name and town fields left at their existing values. Only if no field values were provided would an error message be generated.

Finally, the delete operation simply needs to check that the specified dictionary key exists before attempting to delete it.

File: app.py

```
...

def delete_business(id):
    if id in businesses:
        del businesses[id]
        return make_response( jsonify( {} ), 204)
    else:
        return make_response( jsonify(
             { "error" : "Invalid business ID" } ), 404 )

...
```

Do it now!	Test the operation of the delete operation by trying it with both invalid and valid business **`id`** values.

Try it now!	Update the functions to retrieve, add, edit and delete reviews in the same way **Hint:** Remember that you will need to check for the presence of a valid business **`id`** as well as for a review **`id`**.

4.3 Pagination of Large Datasets

Now that we have a larger collection of sample data to work with, we can resolve another weakness of the API – the delivery of the full set of results in response to a **GET** request to http://localhost:5000/api/v1.0/businesses. When only a small quantity of data is stored, it is probably OK to return the full set to the user in one go, but when 100s, 1000s or even more results are available, it is more usual to paginate them so that only a sub-section of the results are returned at once.

Pagination depends on two parameters – the page number that is requested (i.e. page 1, page 2, etc) and the number of results to be returned on each page. The most flexible approach is to allow these to be determined by the application requesting the resource, so that (for example) a **GET** request to http://localhost:5000/api/v1.0/businesses?pn=2&ps=10 where **`pn`** is the page number variable and **`ps`** is the page size variable denotes that we are being asked for the 2nd page of results (**pn=2**) and that there are 10 results required per page (**ps=10**).

File: app.py

```
...

def show_all_businesses():
    page_num, page_size = 1, 10
    if request.args.get('pn'):
        page_num = int(request.args.get('pn'))
    if request.args.get('ps'):
        page_size = int(request.args.get('ps'))
    page_start = (page_size * (page_num - 1))
    businesses_list = [ {k : v} for k,v in businesses.items() ]
    data_to_return = businesses_list[
                    page_start:page_start + page_size]
    return make_response( jsonify(
                         data_to_return ), 200 )

...
```

In the code presented above, we first set default values for page number and page size and then overwrite these with values in the querystring, if they have been provided. Note the Flask method **`request.args.get()`** to test for the presence of a value in the querystring and to retrieve it. Once the values for page number and page size are known, we can calculate the elements required by the formula

first element = page size * (page number – 1)

For example, if page size is 10, the first element on each page can be calculated as in the following table (remembering that the first page begins with the element at position zero).

Page number	First element
1	10 * (1-1) = 0
2	10 * (2-1) = 10
3	10 * (3-1) = 20
4	10 * (4-1) = 30
5	10 * (5-1) = 40

Therefore, page 1 will consist of elements 0-9, page 2 of elements 10-19, page 3 comprises elements 20-29 and so on.

The final stage is to convert the dictionary into a list so that the required elements can be extracted from it using the slice operation. This is performed by the complex Python **list comprehension** statement

```
businesses_list = [ { k : v } for k, v in businesses.items() ]
```

which can be explained as follows.

Stage 1: We want to return a list called **`businesses_list`**

```
businesses_list = [ ]
```

Stage 2: the list will be a collection of dictionary key:value pairs

```
businesses_list = [ { k : v } ]
```

Stage 3: Each **`{k:v}`** entry will be obtained by looping across the dictionary of businesses

```
businesses_list = [ { k : v } for k, v in businesses.items() ]
```

Finally, the Python slice operation is used to extract the requested elements from the **`businesses_list`** and return those as the response from the request.

Do it now!	Modify the function **`show_all_businesses()`** as illustrated above and test it by specifying different values for **`pn`** (page number) and **`ps`** (page size) in a GET request to API endpoint http://localhost:5000/api/v1.0/businesses?pn=1&ps=10. Note that the order of querystring parameters is insignificant. Check that you receive output such as that shown in Figure 4.4, below.

Figure 4.4 Page 1 and Page 2 of Paginated Output

Try it now!	Implement a similar pagination scheme when retrieving all of the reviews of a business.

Note:	It may feel like a backward step to convert the dictionary of businesses to a list so that the slice operation can be used to extract the required subset of data. This is purely to demonstrate the use of page size and page number variables passed to the API in the querystring. In the coming practicals we will replace the hard-coded data with a database which will enable us to implement the pagination logic within the database query.

4.4 Further Information

- https://docs.python.org/3/library/uuid.html
  UUID Objects – Python manual

- https://www.listendata.com/2019/04/create-dummy-data-in-python.html
  Create dummy data in Python

- https://realpython.com/python-random/
  Generating random data in Python

- https://docs.python.org/3/tutorial/datastructures.html
  Python data structures

- https://www.programiz.com/python-programming/nested-dictionary
  Nested dictionaries in Python

- https://www.datacamp.com/community/tutorials/python-dictionary-comprehension
  Python dictionary comprehension tutorial

- https://documentation.commvault.com/commvault/v11/article?p=45599.htm
  REST API – Response codes and statuses

- https://nordicapis.com/best-practices-api-error-handling/
  Best practices for API error handling

- https://www.pythonforbeginners.com/basics/list-comprehensions-in-python
  List comprehensions in Python

- https://medium.com/better-programming/list-comprehension-in-python-8895a785550b
  List comprehension in Python – creating list from other iterables