COM661 Full Stack Strategies and Development

BE06. MongoDB and Python

Aims

- To present an algorithm for the generation of dummy data for an application
- To introduce the PyMongo package for integration of a MongoDB database with a Python application
- To demonstrate the specification of a database client and selection of a database and collection
- To implement the CRUD operations on MongoDB databases in Python
- To demonstrate the provision of CRUD operations on sub-documents
- To identify opportunities for error trapping within functions.

Table of Contents

6.1 Building a Dataset

So far, we have been working with an API application where the data is hard-coded. Every time the application is re-started, the data is refreshed and all of the changes we have previously made are lost. In this practical, we will see how we can integrate a MongoDB database into the application, creating a persistent data store that is maintained over time.

6.1.1 Generating Dummy Data

In a later practical, we will see how to identify and integrate a real-world data set from an external source, but initially, we will create a set of dummy data related to businesses and reviews to work with.

As a starting point, we will create a new Flask application called **biz** and code a new file ***make_json.py*** to generate the dummy data. This program generates a list of 100 businesses, where each business has a unique *name*, a *town* chosen at random from a set of locations and a random *rating* in the range 1-5. All businesses also have a set of *reviews* that is initialised to an empty list.

File: make_json.py

```
import random, json

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

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

businesses = generate_dummy_data()
fout = open("data.json", "w")
fout.write(json.dumps(businesses))
fout.close()
```

Do it now!	Create a new Flask application ***biz*** containing the Python file ***make_json.py*** as specified above and supplied in the ***BE06 Practical Files***. Run the code by **`python3 make_json.py`** and check that the data generated in the file ***data.json*** represents a valid data set for the application.

Note:	This new application will evolve into the ***Biz Directory*** that is our demonstrator application that we will work with for the remainder of this module.

Now that we have our data set specified as an array of JSON objects in a text file, we can import it into a MongoDB database. Create a new database called ***bizDB*** and import the data into a collection called ***biz*** by issuing the command

```
P:\biz> mongoimport --db bizDB --collection biz --jsonArray
        data.json
```

Do it now!	Import the data file into a collection called ***biz*** in a database called ***bizDB***. Use the Mongo shell to verify that the data has been loaded by generating output such as that illustrated in Figure 6.1, below.

```
test> use bizDB
switched to db bizDB
bizDB> db.biz.find()
[
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    name: 'Biz 1',
    town: 'Banbridge',
    rating: 4,
    reviews: []
  },
  {
    _id: ObjectId('66bc7e83964374e2126146c5'),
    name: 'Biz 15',
    town: 'Belfast',
    rating: 5,
    reviews: []
  },
  {
    _id: ObjectId('66bc7e83964374e2126146c6'),
    name: 'Biz 0',
    town: 'Omagh',
    rating: 2,
    reviews: []
  },
  {
```

Figure 6.1 Importing the Dataset into MongoDB

6.1.2 Preparing PyMongo

With our database created, we can now begin to develop the code that manipulates it. This is most commonly done in Python through a package called **PyMongo** that acts as a driver for a MongoDB database. PyMongo is not native to Python and needs to be installed using Pip by the following command (first ensuring that your virtual environment is running):

```
(venv) P:\biz> pip install pymongo
```

Now, with ***PyMongo*** installed and the database in place, we can create the shell of the database-driven API application. As well as the usual Flask **`import`**s, we also **`import`** the **`pymongo.MongoClient`** class that manages the connection to the database as well as the **`bson.ObjectId`** class that manipulates MongoDB **`objectID`** values.

At the top of the main program body, we create a new database client by calling the **`MongoClient`** class with the URL of the ***mongod*** database server (remember that the IP address 127.0.0.1 is the localhost and that the **mongod** servers listens on port 27017). Next, we select the database to work with and finally identify the collection to which we want to address queries.

Note:	If your application needs to address multiple collections (or even multiple databases), you can repeat these commands as often as you need, specifying different variable names, databases and collections as appropriate.

File: app.py

```
from flask import Flask, request, jsonify, make_response
from pymongo import MongoClient
from bson import ObjectId

app = Flask(__name__)

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB       # select the database
businesses = db.biz     # select the collection

# application functionality will go here

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

Do it now!	Create the shell of ***app.py*** (within your new ***biz*** project) as specified above. You can check that everything is installed correctly by running the application by the command **`python3 app.py`**, though it does not yet have any functionality.

6.2 Building a Database-driven API

With the shell of the application in place, we can now begin to provide the functionality of the API. The best approach to adopt is to copy and paste each route decorator and function from the final version of the ***first*** API (i.e. the state of the application at the end of Practical BE04) – one at a time as we meet them in this practical. The main differences between the BE04 version and the database-driven implementation will be highlighted in bold text in the code that follows.

6.2.1 Get All Businesses

The first API endpoint to develop is that which returns the complete list of businesses, including the pagination introduced in Practical BE04. This code introduces the PyMongo **`find()`** method that returns selected documents from the collection. This method has the form

```
businesses.find().skip(page_start).limit(page_size)
```

where the **`skip()`** and **`limit()`** methods specify the number of documents to be ignored (from the start of the collection) and the maximum number of documents to be returned, respectively. If all documents are required, then the **`skip()`** and **`limit()`** methods can be omitted.

In this example, the **`find()`** method returns all documents (the **`skip`** and **`limit`** specifications notwithstanding), but this can be tailored to search for specific documents and return only specific fields as seen in Practical BE05. For example, to limit the search to businesses from Belfast the command would be

```
businesses.find({town:"Belfast"})
```

while if only certain fields are required, a projection could be added as a second parameter such as

```
businesses.find( {town:"Belfast"}, {name:1} )
```

Note:	In PyMongo **`find()`** projections, **1** and **0** are used to denote **true** and **false** when specifying if a field is to be included or omitted. **1** and **0** are also valid in the MongoDB shell, but are mandatory in PyMongo.

It is important to note that the **`find()`** method returns a cursor (pointer) rather than a set of data values. In the code below, you can see how we iterate across the results returned

by **`find()`** to retrieve each business object and append it to the list of **`data_to_return`**. Note also how we need to convert the **`_id`** value to a string to be able to return it within a JSON structure. We also need to add code to repeat this conversion for the **`_id`** value of every review of the business.

File: app.py

```
...

@app.route("/api/v1.0/businesses", methods=["GET"])
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))

    data_to_return = []
    for business in businesses.find() \
                    .skip(page_start).limit(page_size):
        business['_id'] = str(business['_id'])
        for review in business['reviews']:
            review['_id'] = str(review['_id'])
        data_to_return.append(business)

    return make_response( jsonify(data_to_return), 200 )

...
```

Do it now!	Create the **`show_all_businesses()`** function and test it in the browser or Postman. Try adding values for the **`pn`** and **`ps`** querystring parameters and verify that it works as expected.

Do it now!	Comment out the line that converts the **`_id`** value to a string and re-run the application. Verify that when making a GET request you now obtain an "**Object of type ObjectId is not JSON serializable**" error. Restore the commented line of code to remove the error.

6.2.2 Get One Businesses

To retrieve a single document from a collection, we use the ***PyMongo*** **`find_one()`** method. This takes the same parameters as the **`find()`** method discussed above, but returns only the first document to match the search criteria, or the special Python object **`None`** if no match is found.

Note how we need to use the **`ObjectId()`** method to convert the string parameter to the format required by MongoDB. Also, once again, we need to convert the document's **`_id`** value to a string and repeat the conversion for the **`_id`** field of any reviews.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<string:id>", \
           methods=["GET"])
def show_one_business(id):
    business = businesses.find_one({'_id':ObjectId(id)})
    if business is not None:
        business['_id'] = str(business['_id'])
        for review in business['reviews']:
            review['_id'] = str(review['_id'])
        return make_response( jsonify( business ), 200 )
    else:
        return make_response( jsonify( \
               {"error" : "Invalid business ID"} ), 404 )
...
```

Do it now!	Add the **`show_one_business()`** functionality to the API and test it in the browser or Postman.

Try it now!	Providing a random string for the business ID value in the URL will generate an error that is not caught by our error trapping code, as the call to **`ObjectId()`** expects a valid value to be passed as a parameter. Fix this by adding additional code to ensure that the value passed as the business ID is a 24-character hexadecimal string and return an appropriate response if this is not the case.

6.2.3 Add a new Businesses

Adding a document to a collection is achieved by the **`insert_one()`** method, which takes the document to be added as a JSON formatted parameter. Here, we create the JSON object as a Python dictionary by retrieving the values from the **`request.form`** object as before.

The most interesting thing of note here is the use of the value returned from the ***PyMongo*** **`insert_one()`** method. All PyMongo methods return a value which is an instance of an object related to the method. In the case of **`insert_one()`**, it returns an instance of **`InsertOneResult`** which contains a property **`inserted_id`** which is the **`_id`** key value of the newly-created object.

The guidelines of RESTful API design say that the reply to a POST request should be either a copy of the new resource or a link to it, so here, we use the **`inserted_id`** value to return a link to the new business entry.

File: app.py

```
...

@app.route("/api/v1.0/businesses", methods=["POST"])
def add_business():
    if "name" in request.form and \
       "town" in request.form and
       "rating" in request.form:
        new_business = {
            "name" : request.form["name"],
            "town" : request.form["town"],
            "rating" : request.form["rating"],
            "reviews" : []
        }
        new_business_id = businesses.insert_one(new_business)
        new_business_link = \
           "http://localhost:5000/api/v1.0/businesses/" \
           + str(new_business_id.inserted_id)
        return make_response( jsonify(
                  {"url": new_business_link} ), 201)
    else:
        return make_response( jsonify(
                  {"error":"Missing form data"} ), 404)

...
```

Do it now!	Add the **`add_business()`** functionality to the API and test it in Postman.

6.2.3 Update a Businesses

Document updates are performed by the **`update_one()`** method which takes two parameters – a search object that specifies the object to be updated and a **`$set`** object that describes the fields to be amended. Any fields not specified in the **`$set`** object will retain their current values, hence in this case, the *reviews* for the business are left unchanged.

Again, in this example, we make use of the value returned by the ***PyMongo*** method. This time, the return value is an instance of **`UpdateResult`** which includes a value for **`matched_count`** that is the number of documents in the collection that match the search criteria. Here, we use it to determine whether the business ID value was found and if not, a "bad business ID" response is returned.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<string:id>", methods=["PUT"])
def edit_business(id):
    if "name" in request.form and "town" in request.form and \
       "rating" in request.form:
        result = businesses.update_one( \
          { "_id" : ObjectId(id) }, {
            "$set" : { "name" : request.form["name"],
                       "town" : request.form["town"],
                       "rating" : request.form["rating"]
                     }
          } )
        if result.matched_count == 1:
           edited_business_link = \
           "http://localhost:5000/api/v1.0/businesses/" + id
            return make_response( jsonify(
               { "url":edited_business_link } ), 200)
        else:
            return make_response( jsonify(
               { "error":"Invalid business ID" } ), 404)
    else:
        return make_response( jsonify(
           { "error" : "Missing form data" } ), 404)

...
```

Do it now!	Add the **`edit_business()`** functionality to the API and test it in Postman.

6.2.3 Delete a Businesses

To delete a document from a collection, we use the ***PyMongo*** **`delete_one()`** method, which takes a single search object parameter specifying the document to be deleted. Again, we make use of the value returned from the method and check its **`deleted_count`** property to determine if the search object actually matched a document in the collection. Note that the return code 204 is used to denote that the operation has been successfully completed, but no data is returned.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<string:id>", \
           methods=["DELETE"])
def delete_business(id):
    result = businesses.delete_one( { "_id" : ObjectId(id) } )
    if result.deleted_count == 1:
        return make_response( jsonify( {} ), 204)
    else:
        return make_response( jsonify( \
              { "error" : "Invalid business ID" } ), 404)

...
```

Do it now!	Add the **`edit_business()`** functionality to the API and test it in Postman.

6.3 Working with Sub-documents

The code in the previous section presents implementations for all of the CRUD operations on databases. However, the presence of a collection of review objects for each business gives an extra level of complexity that we must deal with. In this section, we see how to provide add, edit, delete and retrieve functionality for sub-documents in a MongoDB database.

6.3.1 Add a Review

As our database initially contains no reviews, the first action we will consider is the addition of a new review. This is achieved by the ***PyMongo*** **`update_one()`** method, which takes the **`_id`** of the business to which the review belongs as the first parameter and specifies a **`$push`** action on the *reviews* property in the second parameter to add the new review to the collection. As for adding a new business, we return a link to the new review object by constructing a URL with the **`_id`** of the business and the newly-generated **`_id`** of the new review.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<string:id>/reviews", \
           methods=["POST"])
def add_new_review(id):
    new_review = {
        "_id" : ObjectId(),
        "username" : request.form["username"],
        "comment" : request.form["comment"],
        "stars" : request.form["stars"]
    }
    businesses.update_one( { "_id" : ObjectId(id) }, \
                { "$push": { "reviews" : new_review } } )
    new_review_link =
           "http://localhost:5000/api/v1.0/businesses/" \
           + id +"/reviews/" + str(new_review['_id'])
    return make_response( jsonify( \
                { "url" : new_review_link } ), 201 )

...
```

Do it now!	Add the **`add_new_reviews()`** functionality to the API and test it in the Postman.

Try it now!	This code does not check that the business **`_id`** is valid or that all of the required values are provided in **`request.form`** before adding the new review. Implement the error trapping code that checks for this and returns an appropriate error message.

6.3.2 Get All Reviews

The implementation of the function to fetch all the reviews of a business is very similar to that which fetches the details of an individual business. First, the **`find_one()`** method retrieves the business, with a projection that requests that only the reviews element is requested, then a **`for`** loop iterates across the reviews, adding each to the list to be returned after converting its **`_id`** value to a string so that it can be expressed in JSON.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<string:id>/reviews", \
           methods=["GET"])
def fetch_all_reviews(id):
    data_to_return = []
    business = businesses.find_one( \
          { "_id" : ObjectId(id) }, \
          { "reviews" : 1, "_id" : 0 } )
    for review in business["reviews"]:
        review["_id"] = str(review["_id"])
        data_to_return.append(review)
    return make_response( jsonify( \
           data_to_return ), 200 )

...
```

Do it now!	Add the **`fetch_all_reviews()`** functionality to the application and test it in Postman.

Try it now!	This code does not check for a valid business **`_id`** before attempting to return the reviews. Implement the error trapping code that checks for this and returns an appropriate error message.

6.3.3 Get One Review

Fetching a single review is a good example of the power offered by providing each review with its own **`_id`** attribute. As all **`_id`** values are guaranteed to be unique, we can address the review specifically without first fetching the business that the review relates to.

The format of the query is

```
businesses.find_one(
            { "reviews._id" : ObjectId(rid) },
            { "_id" : 0, "reviews.$" : 1 }
)
```

This can be read as "*find the business that contains a review with an _id value matching the review ID passed as a parameter and only return that review*". Note the use of the positional operator **$** which identifies the review that matches the search expression.

When returning the review, we need to remember that the **reviews** element is a list, so even though only a single review is requested, it is returned as a list comprising a single review. Hence, we return it by specifying element **[0]** of the reviews object that is fetched.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<bid>/reviews/<rid>",
           methods=["GET"])
def fetch_one_review(bid, rid):
    business = businesses.find_one(
                  { "reviews._id" : ObjectId(rid) },
                  { "_id" : 0, "reviews.$" : 1 } )
    if business is None:
        return make_response(
          jsonify(
          {"error":"Invalid business ID or review ID"}),404)
    business['reviews'][0]['_id'] =
                     str(business['reviews'][0]['_id'])

    return make_response( jsonify( \
                         business['reviews'][0]), 200)


...
```

Do it now!	Add the **fetch_one_review()** functionality to the application and test it in Postman.

Try it now!	The error trapping code does not distinguish between invalid business **_id** and invalid review **_id**. Rectify this to report a "bad business ID" message if the business **_id** is invalid or a "bad review ID" message if the business **_id** is good but the review _id is invalid.

6.3.4 Update a Review

Updating a review provides another example of the use of the **$** positional operator. Just as when retrieving a review, the review is located by a *find* expression that searches for a business containing a review with an **_id** matching that provided as a parameter. Then, the $set operator is used to update the review with the fields provided in the **edited_review** parameter. Once again, we construct a link to the edited review by using the **_id** values of the business and review and return it as the result of the operation.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<bid>/reviews/<rid>", \
           methods=["PUT"])
def edit_review(bid, rid):
    edited_review = {
        "reviews.$.username" : request.form["username"],
        "reviews.$.comment" : request.form["comment"],
        "reviews.$.stars" : request.form['stars']
    }
    businesses.update_one( \
          { "reviews._id" : ObjectId(rid) }, \
          { "$set" : edited_review } )
    edit_review_url = \
         "http://localhost:5000/api/v1.0/businesses/" + \
         bid  + "/reviews/" + rid
    return make_response( jsonify( \
                  {"url":edit_review_url} ), 200)

...
```

Do it now!	Add the **edit_review()** functionality to the application and test it in Postman.

Try it now!	The code does not contain any error trapping. Add whatever code is required to report on invalid business **_id** and invalid review **_id** values.

6.3.5 Delete a Review

The final element of functionality in the API is that which allows a user to delete a review from a collection. Note that this action uses the **`update_one()`** method (as for adding a new review) as we are effectively modifying the business to which the review relates. Once the business has been located by the *find* expression, we use the **`$pull`** operation to remove the entry with the specified review ID from the *reviews* element.

File: app.py

```
...

@app.route("/api/v1.0/businesses/<bid>/reviews/<id>", \
           methods=["DELETE"])
def delete_review(bid, rid):
    businesses.update_one( \
        { "_id" : ObjectId(bid) }, \
        { "$pull" : { "reviews" : \
                    { "_id" : ObjectId(rid) } } } )
    return make_response( jsonify( {} ), 204)

...
```

Do it now!	Add the **`delete_review()`** functionality to the application and test it in Postman.

Try it now!	The code does not contain any error trapping. Add whatever code is required to report on invalid business **`_id`** and invalid review **`_id`** values.

6.4 Further Information

- https://pymongo.readthedocs.io/en/stable/
  PyMongo documentation

- https://www.w3schools.com/python/python_mongodb_getstarted.asp
  Python MongoDB tutorial

- https://realpython.com/introduction-to-mongodb-and-python/
  Introduction to MongoDB and Python

- https://www.youtube.com/watch?v=3ZS7LEH_XBg
  Connecting to a MongoDB in Flask - YouTube

- https://stackabuse.com/integrating-mongodb-with-flask-using-flask-pymongo
  Integrating MongoDB with Flask using Flask-PyMongo

- https://www.youtube.com/watch?v=4o7C4JMGLe4
  MongoDB operations using Flask-PyMongo - YouTube