Rutgers Course API

Introduction

The **Rutgers Course API** is an attempt to create a server-less and developer friendly version of the Rutgers Schedule of Classes (SOC) API.

There currently exists one API for accessing Rutgers University Schedule of Classes (SOC) Data. This API is not easy to find, is undocumented, and provides only one endpoint.

The currently SOC API does not provide allow for any querying and the data returned contains lots of empty fields, in general is not the easiest to understand, and is some cases unnecessarily duplicates information.

Link to current SOC API Endpoint which provides data for Spring 2018 Undergraduate CS Courses: http://sis.rutgers.edu/oldsoc/courses.json?subject=198&semester=12018&campus=NB&level=UG

This makes it very difficult for Developers to find, understand, and use this API, and provides a tremendous barrier to anyone who wants to use this information to create a program or tool for themselves or for the University.

The aim of my project (currently named the Rutgers Course API), is to overcome all of these issues by providing a publicly available, well documented, and rich versions of the SOC API.

Development Stages

Design

This project has 3 main components:

- Database
- Query Handler
- Request Handler

For the database, MongoDB was chosen because of it's loose and flexible No-SQL collection "schema" and because of it's powerful and easy to use querying system.

Interactions between API endpoints and the database are handler by a Python query module. The query and loader modules using the Pymongo module to interact with MongoDB.

And endpoint requests are handled by either AWS cloud architecture or by a Django web server. The API can easily be transitioned between either of the two request handlers are because they both provide easy mechanisms for interfacing with the Python query module.

PoC

The first step of my work on this project was to complete a very rough proof of concept for the API. This allowed me to get a basic understanding of the technologies I wanted to use.

Below is a break down of the steps taken to complete this stage:

Goal : Recreate current SOC API, with a server-less architecture.

Step 1 - Create Backend DB

Step 2 - Load SOC API JSON's into DB

Step 3 - Write Lambda Logic to query Backend DB

Step 4 - Create API Gateway routes.

Step 5 - Link Lambda Logic to API Gateway in Dev

Step 6 - Correctly Configure API to avoid COORS issue.

Step 7 - Deploy PoC to production.

As of writing this, the endpoint for this PoC API that pushes the same functionality as the current SOC API, exists at the following URL:

https://7cpgmnapaf.execute-api.us-east-1.amazonaws.com/PoC?subject=198&semester=12018&campus=NB&level=UG

With the basic proof of concept completed with the use of the desired technologies, the next step was to design the new database.

All work and source code for this stage can be found in the `Database-Design/` directory.

Database Design

The next step was to design the structure of the back-end database that the new API will query from.

The intent of the API is to allow querying on all collections of data (Professors, Courses, Campuses, Times, etc) while exposing/providing the rest of the data linked to the results of that search.

For example if a user wants to find all 4 credit courses, that user will probably want to see the professors that teach those courses and the campuses where those courses are offered.

But, we want to achieve this accessibility of data without creating lots of duplicates. So in order to achieve this goal we will break all course data into related collections and provide database level links to the related data in other collections.

So at the database level in the **Course** collection: credits, description, title, number are stored as fields, while object id's for the documents that hold the names of the professors for the corresponding courses are stored in __reference__ fields.

This means that this related data is not directly accessible and therefore doesn't need to be stored in multiple locations in the database. Instead if related data needs to be accessed the unique object id can be used to find the record and "expand" the data if the user requests it in their query results.

DB-PoC

The implementation of the "schema" described above was developed in such a way that modifications and updates the any part of the design could be modular and would require modifications to only one file.

In order to achieve this goal, the database loader starts with a JSON config file.

Example:

```
{
    "<collection_name_1>" : {
        "parent_keys" : [
            "<parent_key_1>",
            "<parent_key_2>",
            "<parent_key_3>"
        ],
        "keys" : {
            "<key_name_1>" : {
                "new_key" : "<new_key>",
                "key_mod_method" : "<key_mod_method_name_1>",
                "value_mappings" : {
                    "<SOC_val_1>" : "<NEW_API_val_1>",
                    "<SOC_val_2>" : "<NEW_API_val_2>"
                },
                "augmented_keys" : ["<augmented_key_1>", "<augmented_key_2>"],
                "query_type" : ""
            },
            "<key_name_2>" : {
                "new_key" : "<new_key>",
                "key_mod_method" : "<key_mod_method_name_1>",
                "value_mappings" : {
                    "<SOC_val_1>" : "<NEW_API_val_1>",
                    "<SOC_val_2>" : "<NEW_API_val_2>"
                },
                "augmented_keys" : {
                    "<augmented_key_1>" : "<augmented_key_1_query_type>",
                    "<augmented_key_2>" : "<augmented_key_2_query_type>"
                },
                "query_type" : ""
            }
        }
    }
}
```

The config file is a collection of....collections.

Each Mongo collection is a key in the outer most layer of the JSON. Each of these collections have two things. First, a list of "parent keys", which are the keys, in order, that must be followed from the root of a single SOC JSON. When these parent keys followed in order, they lead to the JSON level where the data that belongs to a collection exists.

The second component of a collection at this level are the collection's keys. Each collection has one or more "keys" which translate to the attributes of the documents that will be stored in the Mongo collection.

Each "key" is itself a JSON Object. Each "key_name" is the original name of the attribute as it is stored in the SOC JSON Object. "new_key" represents the new name of this attribute in Mongo Collection and how it will be queried/reference in the new API.

"value_mappings" allows the original value of the attribute in the SOC JSON to be mapped to a new/different value in the new API. This is used for example, when mapping Campus Codes to Campus Names.

"key_mod_method" is the name of a method inside of course_parsing.py that if implemented will be invoked to modify the attribute value in some other manner that can't directly be translated by a configuration file.

"augmented_keys" serve as a way to map the same attribute to multiple different names. This is used for example, when mapping Campus Codes AND Campus Names.

"query_type" & "augmented_key_query_type" define how the attribute values will be handled when queried from the collection.

This configuration file is parsed by both the database creation and query modules. Updates and additions to the schema and behavior of the API can all be made an immediately reflected by the API just by modifying this single file.

Under this system no changes need to be made to the underlying source code in order to add another collection to the database.

Query-Design

A simple command line query tool was developed to provide the proof of concept for this schema design and configuration system.

Both endpoint handlers, AWS and Django, conveniently parse URL parameters into a python dictionary, where each key is an attribute and the value is the query value.

The param dictionary makes it very easy for the query handler to interpret the desired query.

In order to handle each query type differently, the query handler will "sanatize" (just now writing this report, I'm realizing I spelled it wrong) each parameter based on the query type's behavior.

```python
def sanatize_params(coll_name, params):
    # Santization
    new_params = {}
    for k in params.keys():
        query_type = get_query_type(coll_name, k)
        new_params[k] = get_sanatize_method(query_type)(params[k])
    return new_params

# Find sanatize method in the query module
def get_sanatize_method(key):
    name = "sanatize_" + str(key)
    print("Looking for %s" % (name))
    if name in globals().keys():
        return globals()[name]
    else:
        return sanatize_default

def sanatize_string(data):
    if type(data) != str:
        data = str(data)
    return {"$regex" : data, "$options" : 'i'}
```

The `sanatize_params` iterates over each parameter in the dictionary and looks for a sanatization method for the parameters `query_type`. It does this by calling the `get_sanatize_method` which uses Python `globals()` to look inside `sanatize.py` for a method with the parameter name based to it. If no method `sanatize_parameter` is found, `sanatize_default` is called instead.

The output of `sanatize_params` is a MongoDB query dictionary, so the output of this method can directly be as a query to the Pymongo interface. This is all handled inside of the `query` method inside of `query.py`.

The results of the direct MongoDB query are not ready to be returned as the results of the API call, because the resulting documents will contain ObjectIDs, which should not be exposed to anyone external to the query handler.

```
{
        "_id" : ObjectId("5acfb1a4fd4b6b1647d0dd7f"),
        "name" : "CENTENO",
        "__course__" : [
                ObjectId("5acfb1a4fd4b6b1647d0dd7e"),
                ObjectId("5acfb1affd4b6b1647d0ddad")
        ],
        "__subject__" : [
                ObjectId("5acfb1a0fd4b6b1647d0dd72")
        ],
        "__campus__" : [
                ObjectId("5acfb1a0fd4b6b1647d0dd74")

        ]
}
```

These ObjectIDs are references to documents in other collections that are linked to the data in the resulting documents. There are two options to handle these references: remove or expand them. Removing them is simple by just removing any keys that have names like so: `__name__`, inside of the result dictionary.

The other option is to expand the document data of the documents referenced by the ObjectIDs. This is handled by the `expand` method in `query.py`. Which will query MongoDB for the document with the ObjectID and replace the ObjectID with the non-reference document attributes.

```
{
    "name": "CENTENO",

    "__course__": [
        {
            "number": "112",
            "notes": null,
            "description": null,
            "synopsisUrl": "http://www.cs.rutgers.edu/undergraduate/courses/",
            "title": "DATA STRUCTURES",
            "preReqNotes": "((01:198:111  or 14:332:252 ) and (01:640:135 ))
            <em> OR </em> ((01:198:111  or 14:332:252 ) and (01:640:151 ))
            <em> OR </em> ((01:198:111  or 14:332:252 ) and (01:640:153 ))
            <em> OR </em> ((01:198:111  or 14:332:252 ) and (01:640:191 ))",
            "credits": 4,
            "expandedTitle": null
        },
        {
            "number": "494",
            "notes": null,
            "description": null,
            "synopsisUrl": "http://www.cs.rutgers.edu/undergraduate/courses/",
            "title": "INDEP STUDY COMP SCI",
            "preReqNotes": null,
            "credits": null,
            "expandedTitle": null
        }
    ],
    "__subject__": {
        "number": "198"
    },
    "__campus__": {
        "name": "New Brunswick",
        "code": "NB"
    }
}
```

Once the ObjectIDs are removed or expanded the results dictionary are ready to be returned from the API call to the user.

gateway-poc

This stage was a simple implementation of the query handler behind a Django web server to handle the URL requests.

The completion of this stage provide a final proof of concept for the API.

Conclusion