# What Am I?

The system comprises of a series of RESTful endpoints for performing CRUD operations against segments, families, classes and commodities (as provided in the CSV resources file)

# Installation - Cloud

The system comprises of two components – a Mongo database and a Node.js express-based application. Both are currently installed in the cloud at MongoDB Atlas and Heroku respectively. Thus, the REST endpoints can be readily tested using Postman without having to perform installations if required.

Alternatively it is relatively straightforward to test with local installations.

## Installation Locally

Assuming Node is already installed on a PC, the following is necessary to get the REST endpoints running. From a command window...

git clone <https://git.heroku.com/brownjchallenge.git>

npm install

npm run dev

At this point, it is ready for testing as by default it will connect to the cloud database.

**[UPDATE] – Docker images are now available in DockerHub to install this application locally. It is recommended to use the above git clone command just to observe the code and use Docker to run the application.**

**Please run the following docker commands to get the system running….**

**docker network create demo-network**

**docker container run --name mongo-server --network demo-network -d --rm mongo:latest**

**docker run -p 3000:3000 --name node-server --network demo-network -d --rm jasonbrownuk/noderestapp**

Two containers should now be running. The data storage container is empty and so the first thing to do is load the data, so please execute the following steps…..

**docker logs <container\_id\_of jasonrbrownuk/noderestapp>) -f** to see the logs of the node server outputted to the screen.

In a browser, go to[**http://ip\_address\_of\_docker\_server:3000/load**](http://ip_address_of_docker_server:3000/load)and observe the logs. Wait for ENDING-DATA-LOAD to appear (approx 10-15 seconds)

The system is now ready to test with Postman.

# Architecture

The system stores all resources in a Mongo database – and Mongoose (the Mongo Data Model API) is used in the code to manage interaction with the database. Four node.js mongoose models can be found in the code. Four express routers depend upon the models to manipulate the database. Each router supports the necessary HTTP REST methods for CRUD operations.

The REST endpoints such as creation and update take JSON body payloads and return appropriate HTTP status codes and JSON responses which can be seen in Postman. Some of the APIs take additional parameters (e.g. read) for sorting and paging.

The internal mongoose models support bidirectional referencing of children (e.g. segment to families) and the reverse child to parent relationship (e.g. family to parent segment)

# Configuration Changes

The provided .env files can be used to re-configure settings needed by the application, such as the DB access points.

# Scripts

The package.json file shows 3 scripts can be run…

Run

This is the script that runs on heroku and assumes environment variables have been set up in Heroku (which they have), thus no environment file is needed here. It should not be run from a local Node.js installation.

Dev

This is the script to run when node is locally installed using docker. It uses the dev environment configuration and connects to the cloud database.

Test

This runs a series of unit tests against the user and segment REST APIs. It uses the Jest framework in node.js to simulate a web server and make requests to a simulated CRUD endpoint. It can be used on a local Node.js installation as it is used primarily by developers. It uses the test environment configuration.

Please note I would NOT normally include environment files in github for security reasons, they are included here for ease of distribution.

# Data Loading

The cloud database has been loaded with the data provided in the CSV file. There is a data loading node.js script in the source which can be run from <http://host/load> which performs this task (where host is either <docker-ip-address>:3000 or [brownjchallenge.herokuapp.com](https://brownjchallenge.herokuapp.com/load) depending upon whether node is running locally (docker) or in the cloud). Loading a locally installed DB takes about 15 seconds. Loading the cloud DB takes around 3 minutes depending upon network speed between node and the DB server.

# Testing

The following table shows the endpoints available to be tested using Postman (please note that some Postman exports are included in the github repository which can be imported to save time in setting these up. These have been used to test the application. For each resource (segment, family, class and commodity) it includes creation, deletion, update and 2 read methods.

The URLs below should be prefixed with either https://brownjchallenge.herokuapp.com for the heroku deployed application or [http://<docker-ip-address>:3000](http://localhost:3000/) if deployed locally. So for segments for example….

| HTTP Method | URL | JSON BODY |
| --- | --- | --- |
| GET | /segments/23000000 |  |
| GET | /segments/?sortBy=name&limit=3&skip=6 |  |
| POST | /segments | {"segmentId" : "90000002", "name" : "Zebra Cat"} |
| DELETE | /segments/23000000 |  |
| PATCH | segments/10000000 | {"name": "Industrial Manufacturing and Processing Machinery and Accessories CHANGE"  } |

The same API is supported for the other resources, please change the URL from segments to either families, classes or commodities.

**POST and PATCH considerations in the JSON BODY**

In addition to “name” the following fields are also available in the JSON body for creation and update

|  |  |
| --- | --- |
| For segments… | segmentId  owner |
| For families | familyId  familyowner |
| For classes… | resourceclassId  resourceclassowner |
| For commodities... | commodityclassId  commodityowner |

The ‘<prefix>Id’ fields (e.g. familyId) have values that are the identifiers as provided in the CSV file. The ‘<prefix>owner’ fields (e.g. familyowner) however have values that are an internal database identifier which can be used to change the parent. Examples of values of these identifiers can be found from the response when GETting a resource (e.g "5e9c807484143f298477877f" )

It would be more consistent if the ‘<prefix>owner’ fields in the API were also the identifiers as used in the CSV file, so as to not expose the internal database key – possible improvement to the API.

# Assumptions

The implementations of the APIs currently do not enforce referential integrity so for example deleting a segment may leave orphaned families. Similarly it is possible to create or update child resources without a valid parent (e.g. a commodity without a valid parent class).

Implementations could be changed to those that do preserve referential integrity. For example, delete routines could be designed to fail if there are children (children would have to be deleted first in separate calls to the REST API). Alternatively removing a class could remove all commodities.

Similarly, create and update operations could be changed to ensure valid parents exist first before completing.

Thus 2 versions of the APIs could be offered.

# Improvements

There is a ‘user’ REST API tier and underlying database representation which can be used to manage users for the application. As it stands, no authentication is needed on the Domain data REST endpoints, but this could be easily changed by adding the ‘auth’ middle-ware parameter to each of the domain router methods (as can be seen in user methods in the source code).

This uses JWT tokens to secure and authenticate the user REST endpoints. A user can login using an endpoint (receiving a token) and subsequently provide the JWT token to other endpoints to denote authentication. This is how a client application could interact with the API in an authenticated manner.