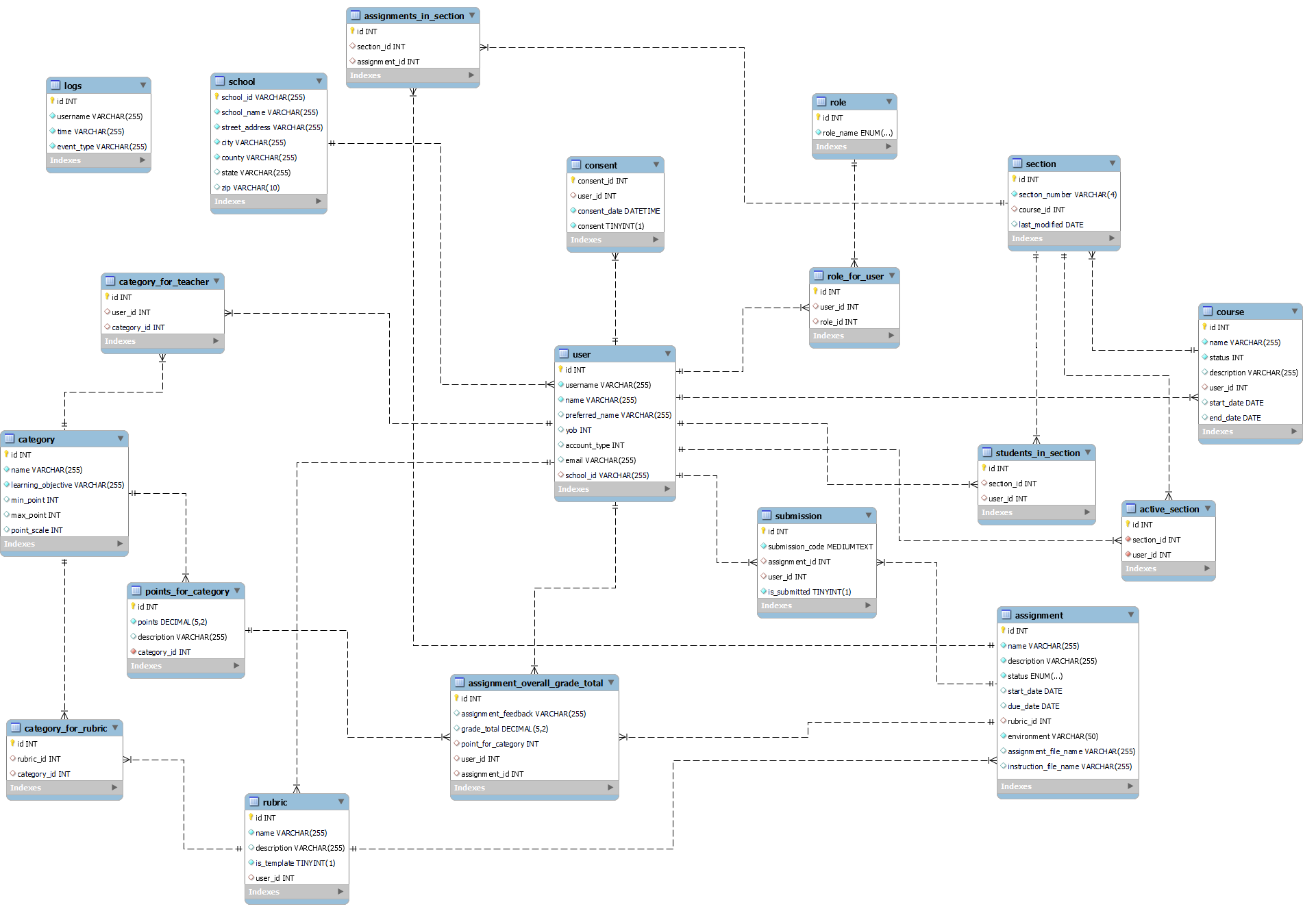
SnapClass Developer’s Guide

## Database Design & Implementation

This section details the design decisions related to the structure of the database for SnapClass, and how each table is related to one another. Many of the entities in the database relate to one another or are dependent on other entities. Each aggregator table serves a singular purpose to map one entity to another. Below, the entity-relationship diagram is discussed along with how MySQL was configured and how the diagram was implemented.

### Database Schema



### Database Definition

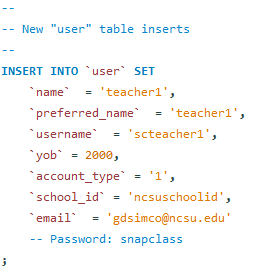
The database for SnapClass is defined using a singular snapclass.sql file which lives in the root directory for the project. The SQL file starts by dropping the snapclass database if it exists and creates a new one. Then each table is defined using the following syntax:



In order to run the SQL file the following command may be used. If the command is run successfully then a fresh snapclass database will be created with the most up to date tables.

|  |
| --- |
| mysql -u root < snapclass.sql |

Throughout the development process mock data was used to consistently be able to use the application. This data is generated using a singular snapclass\_mockdata.sql file which lives in the root directory for the project. The SQL file expects to be run only after a fresh snapclass database is generated and inserts mock data using the following syntax:



In order to run the SQL file the following command may be used. If the command is run successfully then the snapclass database will be updated with data for all existing tables

|  |
| --- |
| mysql -u root < snapclass\_mockdata.sql |

### 

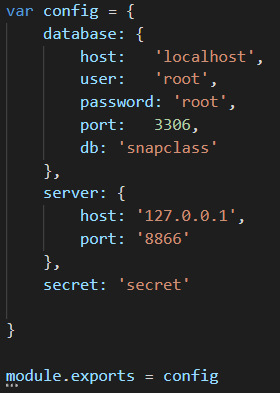
IMPORTANT NOTE: All of the accounts in the mock data use the password “snapclass”.

### Database Connection

The database is contacted using an npm package MySQL-JSON. The module contains functionality to execute database queries through its connection and maintains a list of predefined methods for executing queries. The documentation for the module is located [here](https://www.npmjs.com/package/mysql-json).

The configuration of MySQL-JSON is as follows:

1. Configure Database Variables
   1. File Location: /server/routes/config.js
   2. The database variable contains all the needed information for a secure database connection including the hostname, username, password, port, and database table name
2. Create Database Connection
   1. File Location: /server/routes/db.js
   2. The mysqlJSON variable initializes a new database connection using the configuration variables defined in config.js
   3. The variable is used throughout the backend REST endpoints for executing database queries



## 

## Backend & API Implementation

The backend configuration and logic is contained across many files. Thus, the start of this section defines the low-level design used for the backend followed by a breakdown of each segment.

### Low-Level Design

### 

### Backend Model

**Folder Path:** /server/model/

**File Naming Convention:** Corresponding database entity table

**Component Functionality:** The backend model is responsible for defining the data business logic for each REST API endpoint. Each file correlates to database entity table and contains all database queries related to its corresponding table

**Modules:**

* **MySQL-JSON:** All database queries are executed utilizing mysql-json. The module has predefined methods for common database queries like insertion, updating, and getting by primary key. There is also the ability to create custom database queries using the module.
* **Response Formatter:** The response formatter module was developed to consistently organize how API methods return data and define the content of database error and invalid responses. Each model file requires the response formatter and utilizes its various methods for each callback. The file is located at /server/ResponseFormatter.js

**Example:**

|  |
| --- |
| /\*\* \* Get categories for teacher \*/ exports.getTeacherCategories = function(id, callback) {  db.query(`SELECT category.id, name, learning\_objective, min\_point, max\_point, point\_scale FROM category\_for\_teacher INNER JOIN category ON category.id = category\_for\_teacher.category\_id WHERE user\_id = ${id}`, function (err, response) {  if (err) {  callback(formatter.getDatabaseErrorResponse(err));  }  else {  if (response.length == 0) {  callback(formatter.getInvalidResponse(404, "No categories found"));  }  else {  callback(formatter.getValidResponse({categories: response}));  }  }  }); } |

### Backend Controller

**Folder Path:** /server/routes/\*entity\*/

**File Naming Convention:** Description of API endpoint responsibility

**Component Functionality:** The backend controller is responsible for defining the endpoint for each REST API route. The controller is the first file touched when any API method is called and subsequently calls the backend model to execute any database queries and returns the response as JSON.

**Modules:**

* **Authorization:** The authorization module was developed to restrict access to REST API endpoints based on the user token passed in the request header. The module takes an array of user roles as a parameter and restricts execution of the endpoint based on if the token contains the correct permissions. The file is located at /server/routes/authorization.js

**Example:**

|  |
| --- |
| const express = require('express') const router = express.Router(); const bodyParser = require('body-parser'); const categoryModel = require('../../model/Category'); const auth = require('../authorization'); const roleModel = require('../../model/Role');  // Parsers for POST data router.use(bodyParser.json()); router.use(bodyParser.urlencoded({ extended: false }));  /\*\*  \* Get categories for teacher  \*/ router.get('/:userId/categories', auth.requiredRole([roleModel.enum.TEACHER]), function(req, res) {  categoryModel.getTeacherCategories(req.params.userId, function(value) {  res.status(value.code).json(value.data);  }); });  module.exports = router; |

### 

### Backend View

**Folder Path:** /server/routes/routes.js

**Component Functionality:** The backend view is responsible for configuring the route for each REST endpoint. The view initializes each endpoint with a reference to a controller file and defines the start of each route path.

**Example:**

|  |
| --- |
| **/\*\* \* Add GET categories for teacher route to server \*/ const getTeacherCategories = require('./teachers/get-teacher-categories'); app.use("/api/v1/teachers", getTeacherCategories);** |

### Authorization

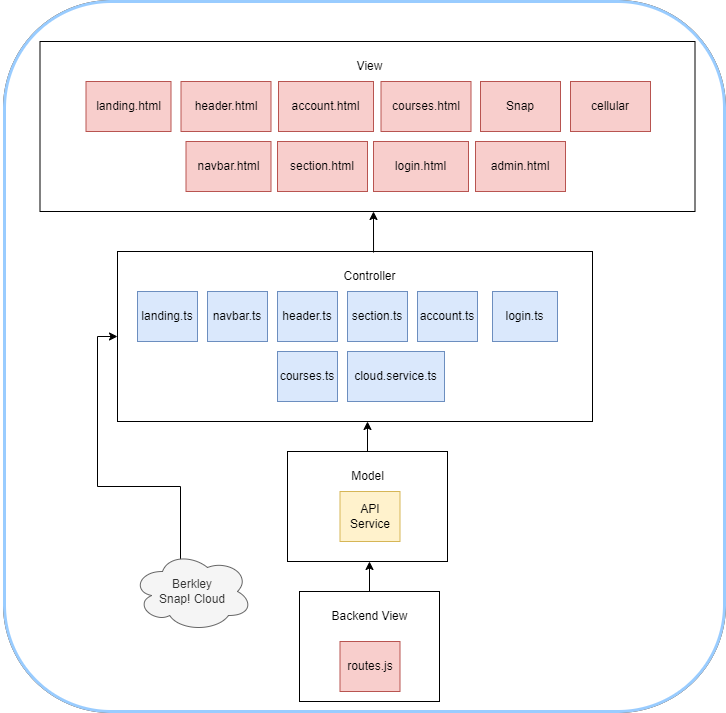
Token generation and authorization is handled by an npm package JSON Web Token. The module contains functionality to generate unique web tokens, marked by a secret key. The process for creating, sending, and receiving a user’s token is outlined as follows:

1. Configure Secret
   1. **File Path:** /server/routes/config.js
   2. The config variable “secret” is used in generating and parsing tokens
2. Generate Token
   1. **File Path:** /server/model/User.js
   2. The login method generates a token using the configured secret
   3. The token contains the user’s information as well as which roles the user has
   4. The token expires in 1 week
   5. The token is returned in the API response for login
3. Store Token
   1. **File Path:** /src/app/services/auth.service.ts
   2. storeUserData() is called on a successfully login API response. The method stores the user token as well as the users information on the client side in local storage
4. Send Token
   1. **File Path:** /src/app/services/token.service.ts
   2. TokenInterceptor retrieves the stored web token from local storage and adds it to the request header for every API call.
5. Authorize Token
   1. **File Path:** /server/routes/authorization.js
   2. requiredRole() serves as a middleware between every API call that requires distinct permissions. The method will decode a web token and compare the permissions encoded to what is required by the REST endpoint. If no token is sent or it has invalid permissions, a 403 Response is returned.
6. Clear Token
   1. **File Path:** /src/app/services/auth.service.ts
   2. logout() is called to clear the local storage of the web token and user information and redirects the user to the guest landing page; requiring them to login again and generate a new web token.

## Frontend Implementation

The frontend configuration and logic is contained across a variety of files based on the structure that Angular. The start of this section defines the architectural pattern used for the frontend followed by a breakdown of each segment, Snap! Integration, services, and finally routing.

### Architectural Pattern



### Frontend Model

**Folder Path:** /src/app/models

**File Naming Convention:** Corresponding entity object

**Component Functionality:** The frontend model is responsible for defining all variables associated with each entity object. For each API response that contains data information representing an entity, the data is transformed into a model object that contains methods for retrieving each attribute of the object. The model also serves to define what the frontend expects from the backend REST endpoints in terms of the content of entity objects since their definitions are disjointed.

**Example:**

|  |
| --- |
| /\*\* \* Section object interface \*/ interface sectionInterface {  /\*\*  \* Section ID  \*/  id: number;    /\*\*  \* Section name  \*/  section\_number: string;   /\*\*  \* ID of corresponding course  \*/  course\_id: number; } /\*\*  \* Implement section object interface  \*/ export class Section implements sectionInterface {  /\*\*  \* Construct dependencies  \* @param data section interface  \*/  constructor(private data: sectionInterface) { }  /\*\*  \* Get section ID  \*/  get id(): number {  return this.data.id;  }  /\*\*  \* Get section name  \*/  get section\_number(): string {  return this.data.section\_number;  }  /\*\*  \* Get course ID for section  \*/  get course\_id(): number {  return this.data.course\_id;  } |

### Frontend Controller

**Folder Path:** /src/app/(teacher/student)/\*component\*/\*component\*.ts

**File Naming Convention:** Corresponding component name

**Component Functionality:** The fronted controller is responsible for defining all dependencies and variables required for the frontend view. Controller files interact with various services and modules to perform actions like API calls, logout, and navigation.

**Example:**

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';  @Component({  selector: 'app-teacher-footer',  templateUrl: './teacher-footer.component.html',  styleUrls: ['./teacher-footer.component.css'] }) export class TeacherFooterComponent implements OnInit {   constructor() { }   ngOnInit() {  }  }** |

### Frontend View

**Folder Path:** /src/app/(teacher/student)/\*component\*/(\*component\*.html/\*component\*.css)

**File Naming Convention:** Corresponding component name

**Component Functionality:** The fronted view is the dynamic display portion of the application. It consists of all the HTML and CSS for each component and dynamically interacts with the frontend controller to get data and execute functions.

**Example:**

|  |
| --- |
| <**div** class="teacher-view">  <**div** class="row bottomBar m-0 py-4 justify-content-center">  <**div** class="logo-container p-0 text-center">  <**img** class="logo h-100" src="/assets/ncstate.png" />  </**div**>  <**div** class="col-1 logo-container p-0 text-center">  <**img** class="logo h-100" src="/assets/bjc.png" />  </**div**>  <**div** class="col-2 logo-container p-0 text-center">  <**img** class="logo h-100" src="/assets/game2learn.png" />  </**div**>  </**div**>  </**div**> |

### Snap! Integration

In order to integrate Snap! into the application, a local instance of the program was added to the project repository. The steps to add Snap! to the project and insert into a view is outlined as follows:

1. Import Snap!
   1. Download Snap! version from <https://snap.berkeley.edu/>
   2. Add Snap! folder to repository root
2. Add Static Path Reference
   1. **File Path:** /server.js
   2. Add reference to Snap! folder to express. By adding the reference, Snap! will run an instance in at the location \*root URL\*/snap.html

|  |
| --- |
| app.use(express.static(path.join(\_\_dirname, 'Snap-4.2.2.9'))); |

1. Import Snap! View with iFrame
   1. **File Path:** Frontend Controller
   2. Use iFrame to reference the location of the Snap! Instance.

|  |
| --- |
| <div id="snapframe">  <button class="snap-fullscreen-toggle" (click)="toggleFullscreen()">Fullscreen</button>  <iframe  class="w-100 h-100"  id="snap-canvas"  [src]="  '/' + (assignment.environment | lowercase) + '.html'  | safe: 'resourceUrl'  "  ></iframe>  </div> |

1. Create JS Methods to Interact with Snap!
   1. **File Path:** /src/assets/js/snap.js
   2. Since all Javascript variables are global, the primary Snap! instance variables “world” and “worldIDE” can be used to execute functionality like saving and uploading Snap! programs.
   3. Most of the method definitions involved in uploading and saving programs is located in \*Snap! Repo\*/src/gui.js

|  |
| --- |
| /\*\* \* Load student submission in snap window \* @param project submission code xml \*/ export function loadSnap(project) {  waitForElement("snap-canvas", function() {  document.getElementById("snap-canvas").contentWindow.worldIDE.openProjectString(project);  }); } |

1. Export JS Methods
   1. **File Path:** /src/assets/js/snap.d.ts
   2. Export JS functions to be used by frontend controllers

|  |
| --- |
| export declare function loadSnap(project); |

1. Import JS Methods in Frontend Controller

|  |
| --- |
| import { loadSnap, newSnap } from '../../../assets/js/snap'; |

Other versions of Snap! such as cellular have been added to the project. The steps to add a version of Snap! to the project and insert it is outlined as follows.

1. Import version Snap!
   1. Obtain clone github repository link of the version of Snap!
   2. Type this git command in your project directory

|  |
| --- |
| git submodule add \*your github repo link\* |

1. Add Static Path Reference
   1. **File Path:** /server.js
   2. Add reference to Snap! folder to express. By adding the reference, Snap! will run an instance in at the location \*root URL\*/snap.html

|  |
| --- |
| app.use(express.static(path.join(\_\_dirname, '\*submodule name\*'))); |

1. Add environment to assignment creation dropdown
   1. **File Path:** snapclass\src\app\teacher\teacher-assignments\add-assignment\add-assignment.component.html
   2. Add the new Snap! environment to the options.

|  |
| --- |
| <select class="col-6" formControlName="environment">  <option value="Snap">Snap</option>  <option value="\*new Snap version\*">\*new snap version\*</option>  </select> |

### Alert Service

**File Path:** /src/app/services/alert.service.ts

**Service Functionality:** The alert service is utilized to display custom error and success messages across the site. Each message is set to display for 5 seconds using a global variable that resides in the alert service. To use the alert service follow these steps:

1. Import AlertService in Frontend Controller
   1. Import service
   2. Add service to constructor

|  |
| --- |
| import { AlertService } from '../../services/alert.service'; |

1. Add Alert to Frontend View by HTML ID

|  |
| --- |
| <**div** class="w-100" id="alert-response"></**div**> |

1. Display Alert in Frontend Controller

|  |
| --- |
| this.alertService.setSuccessHTML(res["message"]); |

### Active Section Service

**File Path:** /src/app/services/active.section.service.ts

**Service Functionality:** The Active Section service is utilized to update and retrieve the active section for the user. When setting the active section, the service retrieves the current user from local storage and updates the section based on user id. When retrieving the active section, the service first gets the current user and then returns the appropriate section.

### API Service

**File Path:** /src/app/services/api.service.ts

**Service Functionality:** The API service is utilized to call all REST endpoints and return standard response data. Any frontend component that requires an API call imports the API service and subscribes to the method call. Once a new API endpoint is defined in the backend, a reference to the endpoint should be added to the API service.

**Example:**

|  |
| --- |
| /\*\*  \* Get point for category  \* @param id point ID  \*/  getPointForCategory(id) {  return this.http.get(`${this.baseUrl}api/v1/points/${id}`)  .pipe(  map(this.extractData),  catchError(this.handleError<any>(`Getting point for category`))  );  } |

### Navbar Service

**File Path:** /src/app/services/navbar.service.ts

**Service Functionality:** The Navbar service provides functionality for toggling the side navigation bar throughout the application. Each component has the ability to close and open the navigation bar.

### Routing Service

**File Path:** /src/app/services/routing.service.ts

**Service Functionality:** The routing service provides the application with a collection of different path names within the system. This allows components to dynamically access path names in given components without knowing the absolute path.

## Back End Testing

**Setting Up and Running Tests**

The back end testing for this project requires several npm packages to run: chai for assertions, Istanbul for coverage, Mocha for the testing framework, and request for making the API requests.

Besides packages, the back end testing requires the database to be in the proper state. After getting the database set up, the database needs to run the file “snapclass\_mockdata.sql”. Unfortunately, our tests have not been configured to clean up properly so each subsequent time the tests are launched, the ‘snapclass’ database needs to be dropped and both ‘snapclass.sql’ and ‘snapclass\_mockdata.sql’ need to be ran again to return the database to its original state.

The tests can be run by navigating to the snapclass project root directory and running the command ‘**npm run testBackEnd**’. This runs a custom script that runs the following command: ‘nyc --reporter=html mocha test/server/routes/testRoutes.js’.

**Overview**

Each API call is categorized into a folder relevant to what it is supposed to test. For example, the api tests relevant to teachers are put in the teacher folder. Inside each of the category folders, each api test gets its own file. This was to cut down on test file size. Additionally, each folder has its own helper JavaScript file with functions to supply data for the tests to use such as the proper request URLs. This way, if a set of API calls changes then the bulk of testing changes would need to be made to the relevant helper files rather than modifying test data that has been hard coded in several times. There is also a global testing helper file called “globalHelper.js” in the root of the test folder that contains some generic tests and some data relevant to testing in its entirety.

In order to keep the mocha tests in separate files and run them as asynchronous tests with mocha handling everything properly a special method had to be implemented. The file ‘testRoutes.js’ is what the testing command actually runs. This file first imports the server file, which launches the server. It then runs a test named “Test Setup” that obtains the login tokens for teacher 1 and teacher 2. When it obtains these tokens, it then runs all of the other tests in the testing folder by importing them and calling their runTest functions, supplying them with the 2 login tokens as parameters. To add new tests, the new testing file should be structured like the other files (with a runTest function that accepts the 2 tokens as arguments) and be imported into ‘testRoutes.js’ using the importTest function. Manually writing code to import each new testing file can be tedious, but it was the only way we found to successfully keep the tests in their own files, run them in whatever order we wanted, and not have to worry about race conditions causing our tests to fail.

After running the tests, the results are printed to the console. The coverage report generated from running the tests can be viewed by opening ‘index.html’ in the coverage folder using a web browser.

**TroubleShooting**

**Unhandled Promise Rejection errors**

Most of the time, this error means the tests were unable to log in successfully. Try dropping the snapclass database and rerunning the ‘snapclass.sql’ and ‘snapclass\_mockdata.sql files.

**2000ms Timeout Errors**

Mocha has a default test timeout of 2000ms. If you are getting this error the first time you have launched/tested Snapclass, it probably means that your computer took too long to get the server running. Simply relaunch the database files and try again.

**Test fails but the database was still modified**

A few of our tests check to confirm whether or not data was added/modified after running certain API calls. One of the tests we had issues with involved checking to make sure that the ‘points for category’ were successfully inside a category after adding that category. It kept failing, but the data was on the database so as far as we could tell it was a race condition where the database had not finished adding the data before the next test accessed it. Consider either just trying the tests again after reinitializing the database and/or adding a wait between the problematic tests.

**Rubric tests fail due to SQL clause**

After troubleshooting the local tests along with rubrics, it seems there is a conflict between the backend and the database. The current sql mode does not allow the `group by` clause which is used during the api call in the backend. To fix this bug, you can update your sql mode by running a query in your database with the following command:

SET GLOBAL sql\_mode=(SELECT REPLACE(@@sql\_mode,'ONLY\_FULL\_GROUP\_BY',''));