SDPM - Going Forward

Note:

Going forward your next team layout should probably be along the lines of 1 backend, 1 canvas integration, and the rest assigned to work on UI. The backend has great functionality and will only need minor tweaks that will only require one person to handle. Canvas integration is a bit difficult and will require much studying of Canvas LTI, but we have the groundwork there. The rest of the group members should focus on Coordinator view and Student views of the UI.

Frontend

Overall, the entire frontend works and we haven't left any bugs in our code (that we could tell at least). There are some overall changes that need to be done and some feature specific changes I'll outline below.

Security

- JWT tokens
  - Currently our JWTs are stored as a token field in our redux state
  - These tokens should instead be stored as browser cookies or within the http local storage.
- All text inputs to the application should be sanitized as well, quiz / peer review / meeting, taking attendance / quiz / peer review.

Quizzes

- We did not get to the ABET tag feature at all, which are certain tags associated with each question for degree accreditation purposes
  - This also includes the "Search Forms" feature which uses those tags as a method of searching for different forms
- Set a timer for how long students have to complete quizzes
- When taking a quiz, you can click the back arrow and leave the quiz to come back and take it as if it's new.
- A way to change a submitted assignments grade for free response questions
- Page to display student overall grades and assignments with their grades
- There are certain fields which should be required such as
  - Quiz title
  - Instance / template choice
    - The associated fields if instance is chosen

        - Actually having questions
            - The questions themselves should have requirements
                - Multiple choice
                    - At least 2 answers and at least one of them being correct
                - Fill in the blank
                    - Should have a correct answer
            - The respective add / create buttons should have handling to make sure these fields are populated

Meetings

- The getinstance for meetings has been created on the backend. We just need a UI to be able to view the attendance of a previous meeting and see who attended

Surveys

- The same applies for meetings as for surveys. We just need a UI to view the previous submitted peer reviews

User History

- When clicking on a student from the class table, there should be a page the coordinator is taken to where they can view the history of that student
    - Submitted peer reviews
    - Submitted quizzes
    - All meetings for that user's team (or another way to handle this, up to you)
- Coordinator should also be able to view the history of that team as a single entity besides the individual students on the team
    - Peer reviews / meetings

User Details

- The endpoint for this is already built. We just didn't get to it in time. But basically this shows overall information about the user (and gives them the option to change password?? idk)
- Upload Roster button available for specific classes needs to be built.

There isn't really much else I can think of besides some quality of life changes. And i can't really think of them off the top of my head considering we're making this document at the end of the semester (sorry, lol). One thing is this though:

- The confirmation dialogs display the class id you chose to assign it to, it would be nice if you could have the class name there instead (the api requires the id though so work that out)

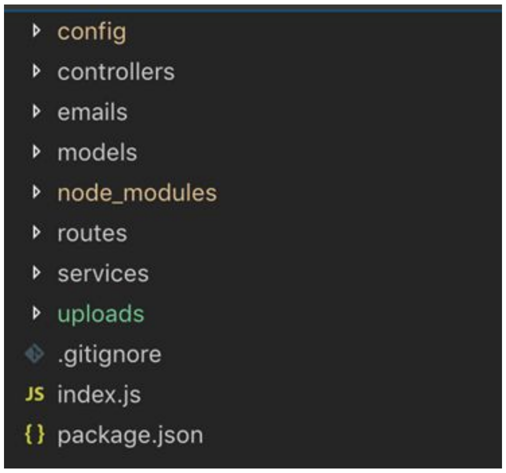
Backend

The backend is also in a completed state. The only things that may need to be added are smaller API functions. The following information will detail the structure of our backend as it stands and what you should do when you add new APIs. We also have the database set up remote in http://phpmyadmin.co/ using the following login information you will have access. Make sure not to make drastic changes when your team is testing!

```
{
  "production": {
    "host": "sql9.freemysqlhosting.net",
    "port": 3306,
    "database": "sql9321404",
    "username": "sql9321404",
    "password": "ZHIlWZPWhY"
  },
```

File Structure:

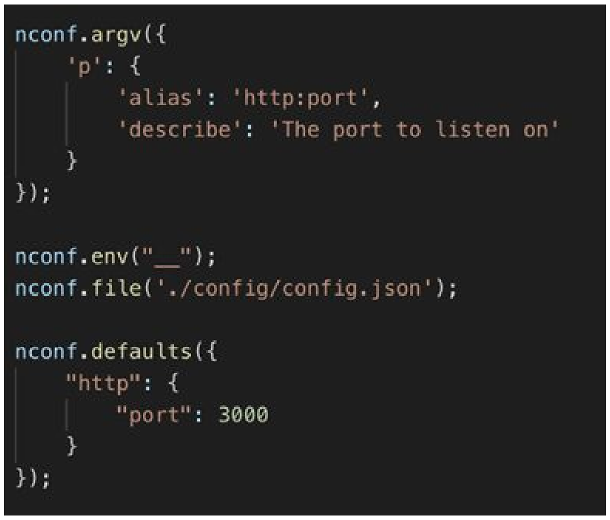
Inside the project's main directory there are two main files and eight subdirectories (minus .gitignore). The backend API's file structure will be as follows:



```
config
controllers
emails
models
node_modules
routes
services
uploads
.gitignore
index.js
package.json
```

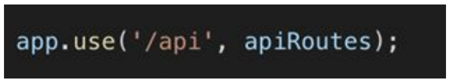
Run Command and Index.js:

The backend is written in Node.js and uses Express as the server. Index.js is the “main” part of the code that handles glueing everything together. If it is the first time starting the server or if you just pulled the code from a repository then, before starting the server, you need to make sure you delete the node modules folder and the package-lock.json file. Then type **“npm install”** to install the dependencies that are currently part of the project. After this you can start up the server by typing **“node index.js”.** This will then start the express server and will display a message in the console, **“Listening on port 3001”**. It will default to running on 3001 but may be configured to any other port and will display a message in the console accordingly. The following section of code is the part that handles all of this:



```
nconf.argv({
    'p': {
        'alias': 'http:port',
        'describe': 'The port to listen on'
    }
});

nconf.env("__");
nconf.file('./config/config.json');

nconf.defaults({
    "http": {
        "port": 3000
    }
});
```

The logic for this is as follows: First, the config.json file is checked to see our port preferences. If there is nothing specified then port 3001 is allocated. At this moment in time, only http is being used for the application but https should probably be considered at a later date.



```
app.use('/api', apiRoutes);
```

This line of code will basically set a prefix of “/api” to all of the route handlers for each individual API call. This also connects all of the routes and allows this index.js file to have access to all of the routes that are available. The variable “apiRoutes” is linked to the api.js file in the routes folder. An example of an api route would be **“http://localhost:3001/api/verifyStudentEmail”**.

Package.json:

This file contains the list of npm dependencies and their respective versions that are being used in the project. When you add an npm dependency to the project, it is important to use “--save” so that it will be saved in the package.json file. This allows it to be automatically installed when running npm install next time a code is pulled.

Node Modules:

This folder contains all the npm node modules that are being used. If it is either the first time ever running the code or if you just did a new pull from the repository, you must delete this directory along with package-lock.json (if it exists) and then type “npm install”. This will reinstall these two. If there is ever any doubt if you need to do an “npm install”, try starting the server. If you get weird errors and the server won’t start, then you should try an “npm install”.

Config Directory:

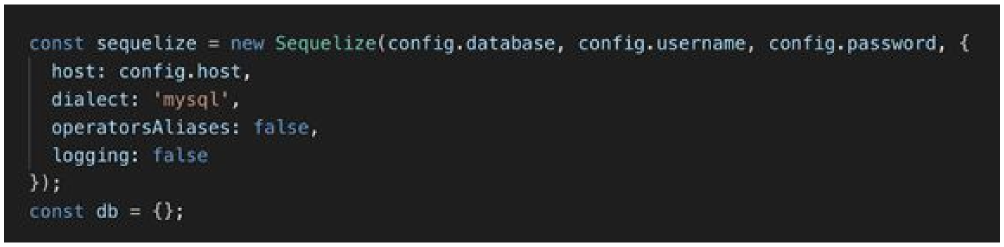
All configuration files will be stored in this directory. It is important to have a separate config directory to be able to keep track of any configurations that are being used or need to be changed. Config.json, at the moment, contains the current database configuration along with info. Config.js contains some pre-set variables that are used for JWT and also for emailing with a test gmail account.

Email Directory:

This folder contains templates (can be used as variables) for email subjects and bodys that can be used wherever necessary.

Models Directory (Index.js):

The models directory contains works with the sequelize dependency to a more structured use and foundation for the database. The only file that is currently in this folder is the index.js file. It is important to note that his index.js file is different from the index.js file previously mentioned that is in the main directory. This index.js file creates a new Sequelize instance that is used to allow MySQL to be used.



```
const sequelize = new Sequelize(config.database, config.username, config.password, {
  host: config.host,
  dialect: 'mysql',
  operatorsAliases: false,
  logging: false
});
const db = {};
```

The first line of code creates a new Sequelize instance that will use the database credentials specified in the config.json file mentioned earlier. Since MySQL is the database type being used for this project, the dialect is set to ‘mysql’.

Controllers Directory:

The controllers directory contains “.js” files for the API requests. Any sort logic or algorithm based code will also be included here. Anything from contacting the database to simple API requests will be in this directory. The files of this directory are separated based on their relationship to its API requests. Each file will contain all the API’s that are relevant to the “file title". So all the student related API will be located in the student.js file and so forth. A controller .js file can contain multiple API's in it. This structure is just a way for things to be grouped for ease of access. The following is a brief summary of each controller currently available and what it should contain (each of these files is a .js file):

- Alerts: Anything that deals with alerts for the coordinator
- Analytics: Anything that deals with analytics. This does not work. I recommend to scrap any existing logic and start this from scratch.
- Classes: Anything dealing with creating or editing a class.
- csvUpload: Contains two separate API's that are used to upload the information in a .csv to a database. One is for inserting students and the other is for inserting teams and adding the students to their respective teams. These can be edited to however you seem fit but overall logic of reading in a .csv and processing its data can stay the same.
- Form: This one is by far the largest file. This contains all the logic to do with forms (quizzes, surveys, etc.)
- frontendTest: This one is just a file that the frontend made used for testing when they need a hardcoded value returned. It can be used however seems fit.
- Login: All the logic used for logging in. This includes JWT token generation and password hashing and comparing. I also included the verifyStudentEmail API in here since it generates a JWT token when it is called and didn't feel like rewriting that logic in a separate controller. The API for changePassword is also here.
- Mailer: This one isn't exactly an API controller (more of a helper function) but I left it in this directory anyway. This function is written so that it can be accessed from anywhere else as long as it is imported. The function takes 3 parameters: toEmail, subject, and body. It will take care of the rest based on the set configuration.
- Register: Contains logic for register along with the setNewPassword API.
- Sponsor: Contains sponsor related things. This was not used in the end, as we decided to handle sponsors like advisors instead of a separate entity.
- Student: Contains student related things
- Survey: Contains survey related things
- Team: Contains team related things

Services Directory:

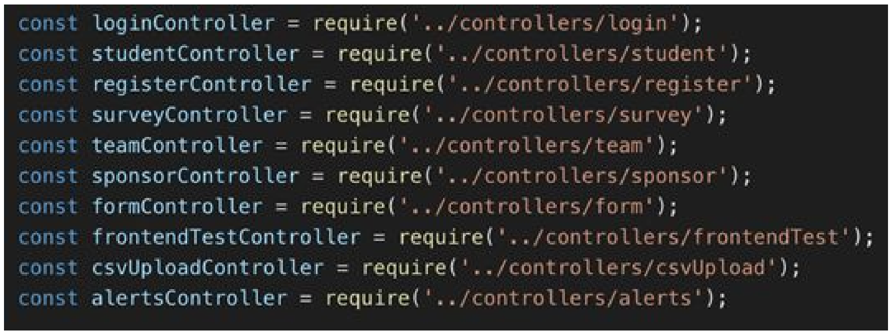
This folder contains middleware for the backend. It currently only contains the passport.js file. Passport helps the JWT token be accessed globally.

Uploads Directory:

This is the folder that any file that is uploaded will go to. It is configured using Multer. The configuration for this is inside of /routes/api.js.

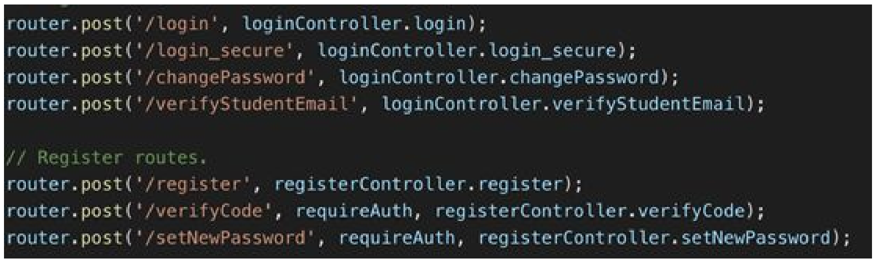
Routes Directory:

The only file in this folder is the file called api.js. Api.js is the file that contains all of the routes in the backend. It is the bridge between the API controllers and the main index.js file. It first contains logic utilizing passport/jwt, express, and multer. It also contains the actual GET and POST request routes for each file and controller. The file utilizes the router module in Express.js to link everything up. At the top of the file are all the respective controllers imported from the controllers directory.



```
const loginController = require('../controllers/login');
const studentController = require('../controllers/student');
const registerController = require('../controllers/register');
const surveyController = require('../controllers/survey');
const teamController = require('../controllers/team');
const sponsorController = require('../controllers/sponsor');
const formController = require('../controllers/form');
const frontendTestController = require('../controllers/frontendTest');
const csvUploadController = require('../controllers/csvUpload');
const alertsController = require('../controllers/alerts');
```

Then, the actual route handling is listed. Most, if not all of the routes are POST requests. Each of the requests uses its respective controller to access the specific API call in order to perform the request.

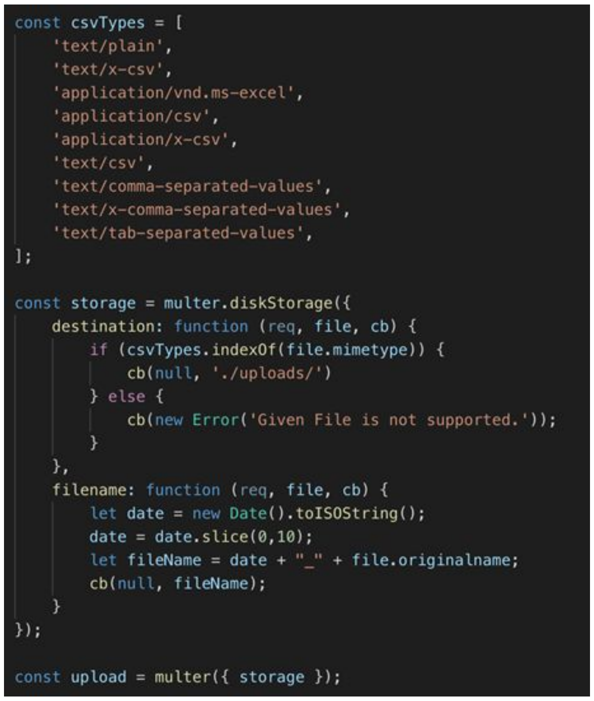


```
router.post('/login', loginController.login);
router.post('/login_secure', loginController.login_secure);
router.post('/changePassword', loginController.changePassword);
router.post('/verifyStudentEmail', loginController.verifyStudentEmail);

// Register routes.
router.post('/register', registerController.register);
router.post('/verifyCode', requireAuth, registerController.verifyCode);
router.post('/setNewPassword', requireAuth, registerController.setNewPassword);
```

If an API should include a JWT token when used, you should have the “requireAuth” variable. If it is not there, then that API will not need a JWT token when called.

Multer:

The api.js file also contains the logic used to upload csv files. It utilizes the library called Multer. Right now the code is set to take in a file, check if it is a csv, then rename it and add it to the uploads folder.



```
const csvTypes = [
    'text/plain',
    'text/x-csv',
    'application/vnd.ms-excel',
    'application/csv',
    'application/x-csv',
    'text/csv',
    'text/comma-separated-values',
    'text/x-comma-separated-values',
    'text/tab-separated-values',
];

const storage = multer.diskStorage({
    destination: function (req, file, cb) {
        if (csvTypes.indexOf(file.mimetype)) {
            cb(null, './uploads/')
        } else {
            cb(new Error('Given File is not supported.'));
        }
    },
    filename: function (req, file, cb) {
        let date = new Date().toISOString();
        date = date.slice(0,10);
        let fileName = date + "_" + file.originalname;
        cb(null, fileName);
    }
});

const upload = multer({ storage });
```

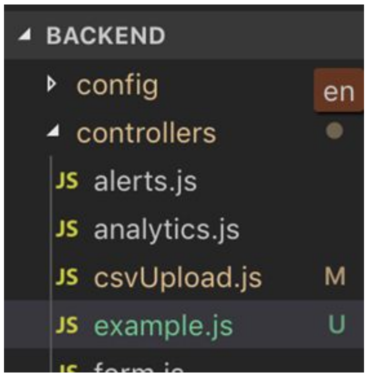
Module.exports:

At the end of any file that contains something that needs to be accessed from another file, there is a line of code that is “module.exports” that allows the contents specified to be exported to and accessed from another file.

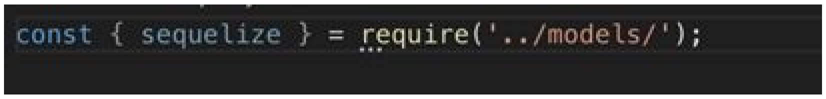
How to Create a New API Route:

Creating a new route is pretty simple. First you must choose whether you will edit an existing controller file, or to create a new one. For example’s sake, I’ll be starting from scratch.

1. Go to the controllers directory and create a new .js file of the controller. I’ve created a new file called example.js here:

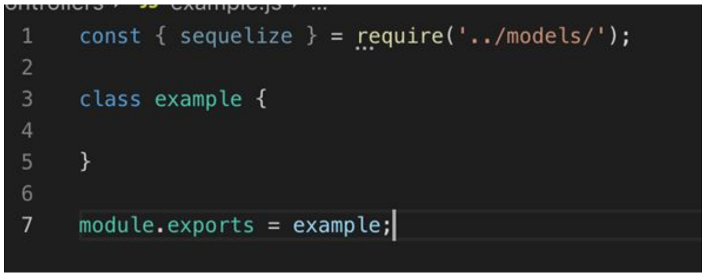


2. Now open the blank file to start adding code to it.
3. If any of the API routes that will be created in this controller need to access to the database, then be sure to import sequelize. By including sequelize as pictured in the following, it will take care of the config as specified in config.json and access the database accordingly.



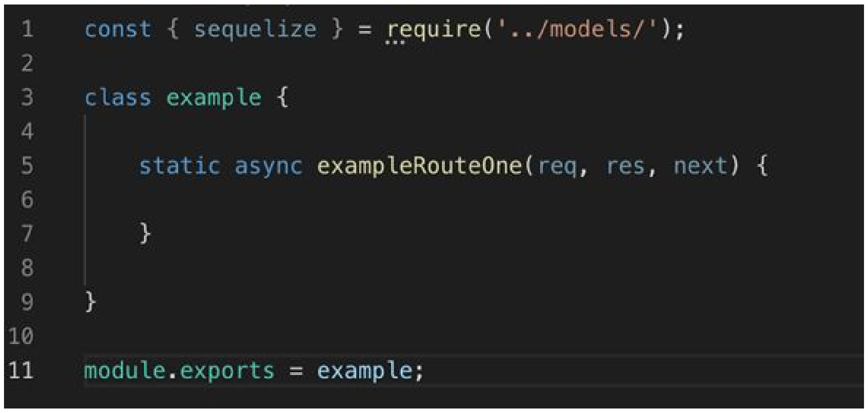
```
const { sequelize } = require('../models/');
```

4. Next is to create a class and then at the end add module.exports so that the class can be exported. The file should currently look like this:



```
const { sequelize } = require('../models/');

class example {

}

module.exports = example;
```

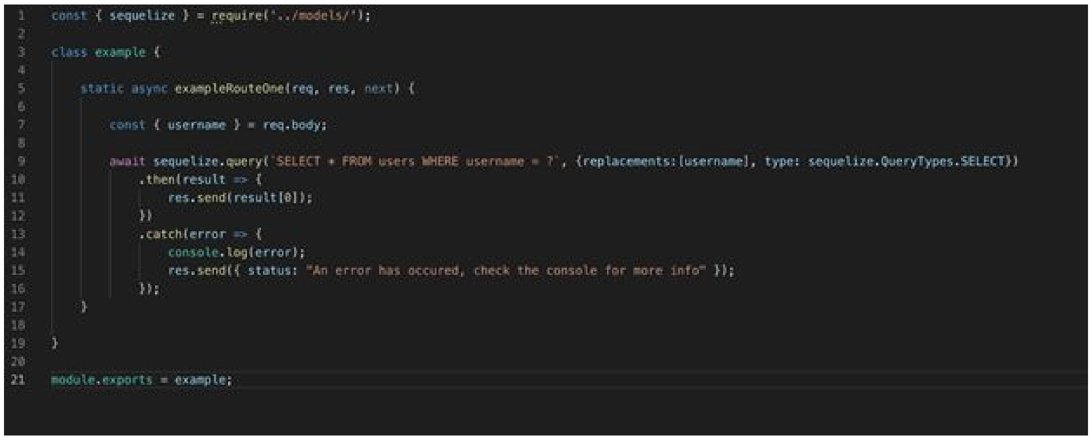
5. Now we can create a route. I will call this route exampleRouteOne. It is important to include “static async” along with the parameters “res, req, next” in the route function. These are standard for express route handling and async database connections:



```
const { sequelize } = require('../models/');

class example {

    static async exampleRouteOne(req, res, next) {

    }

}

module.exports = example;
```

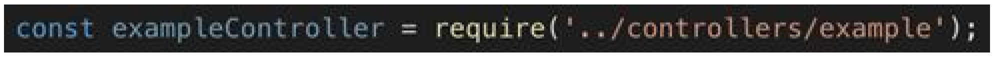
6. Note: I can make as many static functions as i want in a controller as long as they are written properly. But the whole point of breaking them up is to be able to categorize things.
7. Now it is time to write whatever logic we need. For this example, I’ll just take in the request body and then do a simple select statement from the database and return its results as

an Express response. In this case I am going to take in a username, search for it in the database, and finally return all of its associated information as a response while including a catch-block to catch for any error that may occur:



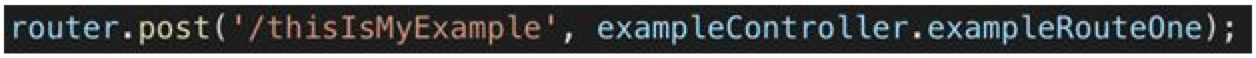
```
const { sequelize } = require('../models/');

class example {

    static async exampleRouteOne(req, res, next) {

        const { username } = req.body;

        await sequelize.query(`SELECT * FROM users WHERE username = ?`, {replacements:[username], type: sequelize.QueryTypes.SELECT})
            .then(result => {
                res.send(result[0]);
            })
            .catch(error => {
                console.log(error);
                res.send({ status: "An error has occured, check the console for more info" });
            });
    }

}

module.exports = example;
```

8. Now that a controller is created, it must be added to the list of routes in /routes/api.js
9. This next set of instructions is all done in /routes/api.js
10. Open up the api.js file in the directory and add import the newly created controller as so:



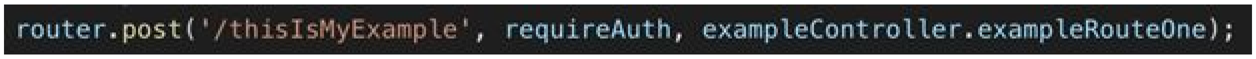
```
const exampleController = require('../controllers/example');
```

11. The naming scheme currently used right now is the name of the file + Controller. And then use require to import it. Note: You don't need include ".js" when importing a .js file since it will automatically assume it is a .js file by default.
12. Now that we have imported the controller, it is time to create the actual GET/POST request.
13. In the same api.js file, scroll down and add the following code (I will break down what exactly is happening in following steps):



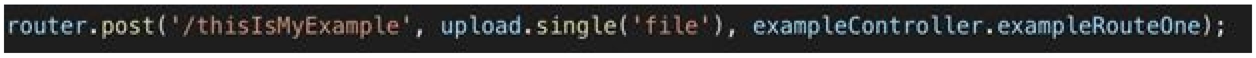
```
router.post('/thisIsMyExample', exampleController.exampleRouteOne);
```

14. The above screenshot is an image of a POST request. If you want the API you just created to be a GET request, then replace "post" with "get" in that line of code.
15. The part in the orange is the name of the api. This is what the frontend will type when making this api call. For this specific request, the call would be "http://localhost:3001/api/thisIsMyExample"
16. The last part in this line of code is to bridge the connection by including the controller the we imported along with the name of the specific API that this route will use.
17. Aside from a couple exceptions, thats it! Save the code and this API is ready to be tested.
18. This specific route that we created is a POST request that takes in the parameter "username" in the form of JSON. It takes the username and returns all the relevant info for that user.
19. This route is NOT protected with JWT. In order to make it so that a JWT token must be included as a header, add "requireAuth" to the route as so:

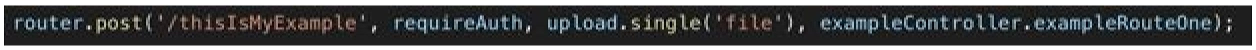


```
router.post('/thisIsMyExample', requireAuth, exampleController.exampleRouteOne);
```

20. If you need the route to accept a file upload, the following images show how to do that for unprotected and JWT-protected routes:



```
router.post('/thisIsMyExample', upload.single('file'), exampleController.exampleRouteOne);
```



```
router.post('/thisIsMyExample', requireAuth, upload.single('file'), exampleController.exampleRouteOne);
```

21. If you are uploading a file and testing it in Postman, be sure to include the “key” as “file”, or it will not work.
22. The code for how to take in a file along with a body through an express request can be seen in the csvUpload controller.

That should be all you need to make sense for how the backend code is structured and used.

Canvas

The first thing involving Canvas testing is getting an instance running. To do this, one of our programmers used a local Ubuntu VM. In that VM, install docker. After that, follow the instructions detailed in our final doc to install and run the container.
*NOTE: your host machine can also access this Canvas instance, all you have to do is open your host machine browser and go to this link: ‘[IP address of your VM]:3000’.

Canvas implementation, overall, is about halfway complete. There is a folder in the backend folder called ‘canvas’. This folder holds the two files that have most of the code involving communication with Canvas: lti.js and canvasapi.js.

LTI

If needed, study these links to better understand how LTI works:
https://www.imsglobal.org/basic-overview-how-lti-works
https://canvas.instructure.com/doc/api/file.lti_dev_key_config.html

For LTI authentication, the backend server has been configured to take POST requests; this can be seen in the ‘index.js’ file in the backend folder. ‘lti.js’ holds the code involving a Canvas LTI launch. This implementation is based on this project: https://github.com/js-kyle/nodejs-lti-provider. We first tested/studied Canvas’ LTI by implementing this project. After it worked, we essentially copied most of its code for our implementation, just tweaking parts of it.

Unfortunately, our implementation of LTI is only able to successfully recognize and review the data from an LTI launch (it prints out some of the launch info on the console window). It also redirects that user to the (assuming its running) frontend login page. It does not sign them in yet. We do not have any implementation checking a user’s data to see if they exist in our system (this would be the job of a future group). The system does reject a POST request in two situations: if the POST request did not come from a consumer, like Canvas, or if the POST request's consumer key is not recognized in our system (we only have one combination of a key/secret at the moment: **consumerkey**/**consumersecret**).

To test out an LTI tool in Canvas- please check 13.2.1 of our Final Documentation.

Canvas API

Studying this document is helpful for understanding how to make Canvas API calls and to add more: https://canvas.instructure.com/doc/api/

For the Canvas API, we utilized the 'request' Javascript module. 'Canvasapi.js' holds code for functions to make GET requests to the API. An example testing script exists in the backend called 'canvastest.js'. You can run this test using the command 'node canvastest.js'

'canvasapi.js' has four functions, all tested and confirmed to work on the aforementioned Canvas instance. The format of each function is essentially the same, the main difference is the link: links need to be modified to get exactly what is desired from the Canvas API (course roster, group members, a user's courses, etc). Each function's second parameter is to get some kind of ID to specify a course/group/user.

Each function's first parameter is a callback function: this should be a single argument function that must be given to process the data given from the request; maybe to use that data to update some field or just to be stored in a variable, etc. Callback functions are expected to run immediately after the system receives the response from the request. Please look at the code in 'canvastest.js' to have a basic understanding on how a callback function should look and how it displays/accesses the data.

For future plans, the main goal for the next group should be fully implementing API calls; binding these functions to frontend buttons. Future groups should feel free to use a different/better implementation (if they can find one) to this 'request' module implementation. We really just focused on having some kind of implementation that actually works; optimization would be the work of some future group.

Necessary Credentials

The user logins are in the email address. All other logins will be posted here.

Associated Email

Gmail:
Email: sd.coordinator2020@gmail.com
Password: coordinator221

Remote Database

phpmyadmin.co

Server: sql9.freemysqlhosting.net
Name: sql9321404
Username: sql9321404
Password: ZHIIWZPWhY
Port number: 3306

Administrator Application Login

Username: admin
Password: 123

ACCESSING VM:

1. CISCO if you're not on campus
ucfvpn-1.vpn.ucf.edu
email and password

Old VM
2. Filezilla or Terminal (Mac)
IP: 10.171.204.179
user: student
password: orange

New VM
username: student
password: password
IP: 10.171.204.184

If on Mac Terminal: ssh student@IP_ADDRESS
Enter password