Tech Talent Pipeline Bootcamp and Internship Technical Review by Jayna Menard

Purpose: In-depth review on final project created in the TTP Winter Bootcamp.

Worked on: Backend and CSS.

**Winter Bootcamp**

**Personal Notes**

The Tech Talent Pipeline Winter Bootcamp was created for selected students to be heavily prepared for any task at our selected internships. This course teaches us Git practice using Visual Studio and React, terminal commands, HTML, CSS, and JavaScript. It was an enriching experience and I hope many other will have the opportunity to attend this bootcamp. The most striking thing about the bootcamp is that it is only for a month. In that month, we must learn a variety of things without break. As an aspiring computer scientist, I was very excited to begin learning. The only thing that was difficult was managing my time to keep up with new subjects while doing heavy duty projects on old subjects. But the endless nights without sleep were worth it. I grew into a better computer scientist and with the knowledge and this knowledge to complete the final assignment of the course, the Campus Manager. This was the most fruitful project we coded in the bootcamp.

**Final Project: Campus Manager in React**

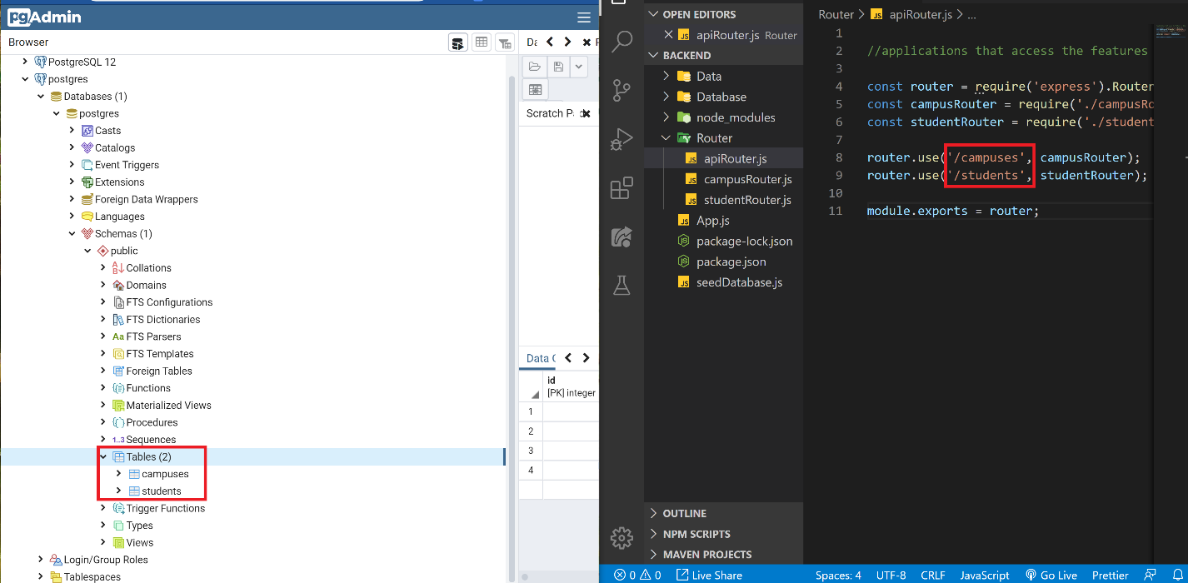
Our professor assigned a project that should resemble an administration page for local campuses (i.e. CUNYfirst). Below are the required actions for a User when connecting to the webpage. The requirements highlighted in red were not implemented in the project due to lack of time.

As a User, I:

* [ ] will land on a visually pleasing homepage by default, which allows navigation to view **all campuses** and **all students**
* can navigate to **all campuses view**, and
  + [ ] see a list of all campuses in the database
  + [ ] see an informative message if no campuses exist
  + [ ] add a new campus
    - [ ] with a validated form displaying real-time error messages
    - [ ] which redirects to the new campus’ **single campus view**
* can navigate to a **single campus view**, and
  + [ ] see details about a single campus, including enrolled students (if any)
  + [ ] see an informative message if no students are enrolled at that campus
  + [ ] navigate to any student’s **single student view**
  + [ ] delete the campus (and elegantly handle associated students)
  + [ ] edit campus information (including adding/removing students)
    - [ ] with a validated form displaying real-time error messages
    - [ ] which redirects back to the **single campus view**
* can navigate to **all students view**, and
  + [ ] see a list of all students in the database
  + [ ] see an informative message if no students exist
  + [ ] add a new student
    - [ ] with a validated form displaying real-time error messages
    - [ ] which redirects the new student’s **single student view**
* can navigate to a **single student view**, and
  + [ ] see details about a single student, including the campus at which they are enrolled (if exists)
  + [ ] see an informative message if student is not enrolled at a campus
  + [ ] navigate to **single campus view** of the student’s enrolled campus
  + [ ] delete the student
  + [ ] edit the student’s information (including campus s/he is enrolled at)
    - [ ] with a validated form displaying real-time error messages
    - [ ] which redirects back to the **single student view**

Starting off, we must understand how the campus manager works for the client. Before a client can connect to the webpage, a server must be running. The server provides an internet connection to a port and the handles database queries associated with the webpage. As the server is running, it waits for the client to connect to the same port. For our group project, we used Node.js, Express, and Sequelize to do so. Under the Router Folder of our back-end, we have three JavaScript files – apiRouter.js, campusRouter.js, and studentRouter.js.

apiRouter.js uses applications that access the features of the database. Using Express, we connect a new client to the PostgreSQL server and define our routes (campusRouter.js, studentRouter.js) that are defined as tables in our database (campuses, students).



For both campusRouter.js and studentRouter.js, async/await code are written in Node.js. This code was taken from <https://node-postgres.com/>, which allows our code to do HTTP requests such as getting data (.get), uploading data (.post), and deleting data. Our main issue was making sure that we were able to record data entered by the client into the database. Initially, using Postgres was very simple. By using the SQLShell, we created tables, added data, and deleted data as we pleased. However, for this project, data added on by multiple clients was not saved. The issue was making sure that the information entered was unique.

The campuses table consisted of 4 columns - name(String), bio(String), address(String), and img(String). For the students table, we have name(String), img, and gpa(Integer). Our students and campus are visible when hard-coding test models (campus.js and Students.js). However, when the client added information, the front-end will show the newly added student/campus, but not in the database. What we needed was a **primary key**. A **primary key** is a constraint that is used to identify each record uniquely in a table. Using an **id** for the campuses table, the students table can use that id as a **foreign key**, which links the two tables together. We then added an **id** column for the campuses table and a **campusid** column for the students table to achieve the results . For a better understanding, here is a sentence and image describing the relation between both tables (relation is also defined in the Assocition.js file):

“A student belongs to a campus; A campus has many students.”

A close up of text on a white background

Description automatically generated

\*Gold key is the primary key; Silver key is the foreign key.

Now, instead of directly declaring the columns in the database, the columns of the table are defined in the code (Campuses.js, Students.js). This makes it easy to create models and import them correctly into our database and application (index.js). Then to check if the data is logged, we used Sequelize to log in to the database server with the appropriate credentials (db.js).

A few functions are added to make sure the info added populate the tables in order, place our dummy student data in random campuses which is useful to see what kind of data we want to populate the database with in the future (seedDatabase.js). Finally, Express is again used to connect to the designated port (3001) through our React router. We also imported the body-parser dependency as middleware to parse the JSON body text into URL-encoded data, which is how the browser interprets and sends data between POST requests.

Moving on to the front end of the project, we add CSS files and routes to make sure each page is connected. For each page, a JavaScript file is added so any actions made by the client will trigger the back-end HTTP requests by connecting each page to the specific route URL created by us in async/await code and to warn us in the terminal of any errors that may occur (i.e. bad internet connection leading to data not being saved, database server not connected).

To run the application, follow these steps:

1. Visit https://github.com/JaynaMMenard/due-01-19-2020-full-stack-crud-application-floreo-labs.git for the complete project.
2. Open pgAdmin and enter the master password to connect to the Postgres database server of your choosing.
3. Change the code in **db.js** to your own Postgres credentials.

­­ A screenshot of a cell phone

Description automatically generated

1. Open the “Backend” folder in Visual Studio code. Upon loading, open the integrated terminal and type in “npm install” to install the necessary React packages.
2. After installation, in the terminal, run “nodemon App.js”. This event will allow the backend to connect to the server and we will see real-time changes in the terminal when the webpage data is edited (front-end).
3. Now, open in a separate VS window the “FrontEnd1”. Upon loading, open the integrated terminal and type in “npm install body-parser, express, morgan pg, websocket-driver” to install the necessary React packages.
4. After installation, in the terminal, run “npm start”. The React code will open the webpage on port 3000.