Will Gillette, Jess Sokolski, Kacey La, Ryan Fitzgerald, Matt Quigley

Dr. Mongan

CS-375: Software Engineering

21 February 2023

Academic Planner Design Report

To reiterate from previous reports, the primary goal of this project is to optimize and reorganize the current degree planning features within the Student Access Portal. In our requirements reports, we discussed our decision to partition the application into three primary acts: user authentication, degree progress, and degree planning. These acts comprise numerous user requirements, and we wish to complete those of which are specified under our Minimum Viable Project Scope (MVP).

To implement user authentication, we must develop a token system that handles "sessions", the functionality that allows users to log in and out of their accounts. Additionally, we must implement a SQL database structure on the backend to determine whether users can sign in and if they can create an account based on their specified fields. This database will also enable us to retrieve user data from the server and replicate it on the view. Finally, we must ensure that only authenticated users can access the pages pertaining to the other acts: the degree progress and degree planning pages. Due to this, the other acts are extremely dependent on user authentication.

The degree progress page will likely account for the bulk of our overall timeline because we plan to renovate Ursinus' current system due to its many issues. Our first requirement with this page will be to display user information on the view by receiving it from the server. This information consists of the individual's GPA, major, anticipated graduation date, and headshot.

In addition, we will store the list of courses in addition to other global state information in our Redux store, which will help us with displaying course information. We will then create a React component that will serve as a placeholder for core requirements, allowing us to display course information for each of the core requirements a student has fulfilled. This component will have a property that contains course information, allowing us to load and change courses that students assign to core requirements. To change the course that fulfills a core requirement, we will include a dropdown menu of valid courses and perform validation checks to ensure that the course is eligible for assignment. Following this selection, we will send a request to the server and update the database, receiving a response about whether this process was successful and updating the view accordingly. Finally, we will accumulate the credits that a user has taken and display it via a progress bar component. Due to this complex functionality, we foresee this aspect of our application taking the majority of our time.

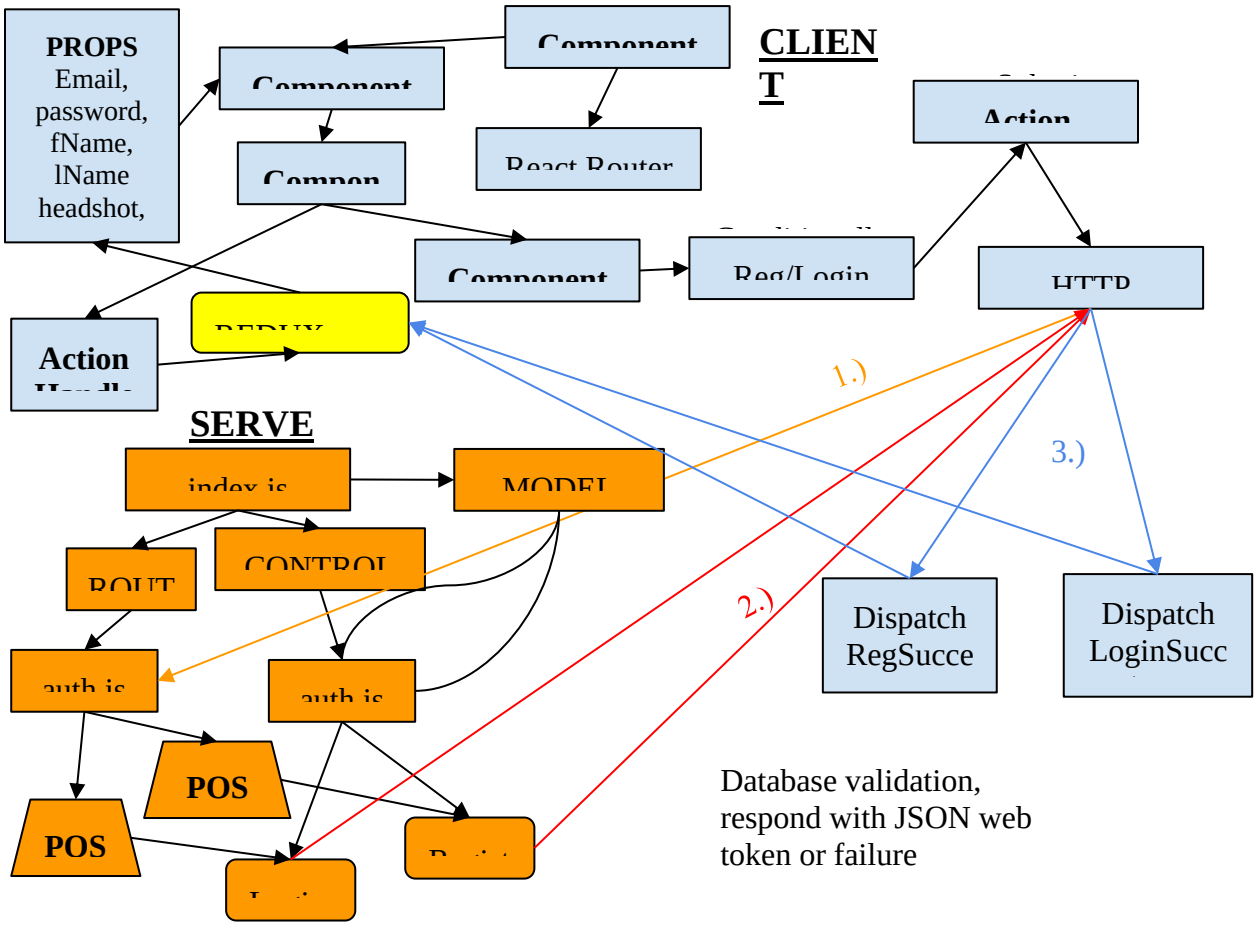
We will dedicate the final segment of our project to replicating and finetuning Ursinus' degree planning page. Its most significant user requirement includes implementing the ability to filter the course catalog so that users can effectively assign courses to certain semesters. The available semesters to which users can assign courses are dependent on a user's anticipated graduation date, which we can fetch from the Redux store. Our plan is to develop a "semester component" and a "course information component." The "semester component" will include the semester's name and credit total and essentially act as a container for "course information components." A "course information component" will include properties about the course such as any pre-requisites and core categories it fulfills. There will be a dropdown menu that allows users to switch to another available semester, causing the semester component to re-render, creating "course information components" for the courses that the user has assigned to the

specified semester. Upon assigning a course to a semester, we plan to make a request to the server to validate that the user is eligible to take the course and has not taken it already. In addition, the server will check the credit total of the respective semester to ensure that a user has not planned over sixteen credits. Following these checks, the client will receive a response from the server indicating whether the course can be added to the respective semester and dispatch an action to the Redux store to update the view accordingly. As per course browsing and filtering, like the current model, we plan to include a form dedicated to that. Users will be able to filter out "even/odd year courses", courses fulfilling a certain core category, and courses that users are eligible to take based on fulfilled prerequisites. This page will operate similarly to the model described above; we will have a container component that stores course information components. However, users will be able to select a semester in which to assign the course via a dropdown menu that respects their anticipated graduation date. Upon doing this, the server-side communication process described above will occur. Below is a compact table of the requirements described above:

User Authentication	Degree Progress	Degree Planner
1. Home page a. Header component → Login/Registration Form 2. Token System implementation to handle "sessions" 3. SQL database for users that holds registration data a. API endpoints that lead to validation checks for registration using the database 4. Set up React router → add	1. Receive user summary data (GPA, major, graduation date, headshot) from server → Display it on the view 2. Set up Redux store → fetch all course data from the server and store it locally in this store 3. Core requirement container component a. Replicate this for each of the core requirements b. Include a property	1. Create a button component that toggles the course browser component (modal?) a. Include a search bar that allows users to browse for courses (display first twenty by default) b. Create a container for "course information components" i. Create a "course information component" and replicate it for each course currently displayed 1. The course information

protection to routes so that only users with tokens can access them	for course information, displaying it on the view c. Construct a course dropdown menu sub-component, displaying eligible courses for that core category (may change to a modal form) → Send a request to the server and re-render the container component if necessary 4. Create a progress bar component a. Compute the total amount of credits a user has completed and display it within this component 5. On the server, create an API endpoint that performs validation checks after a user has selected a course to fill a core requirement	component will have a checkbox that allows users to select the course c. Include various filters above the search bar i. Re-render the container component after filters have been toggled d. Include a dropdown menu to select the semester in which to add the course e. Include a submit button that performs validation checks on the server, updates the Redux store, and re-renders the "semester component" if necessary 2. Create a semester component a. Fetch data from the server about what courses the user has planned under that semester b. Display these courses in "course information" sub-components within the semester component i. Within each sub-component, include a button to remove the course from the semester currently displayed c. Include a dropdown menu to switch the semester plan currently being displayed 3. On the server, create API endpoints for adding and removing courses from semesters

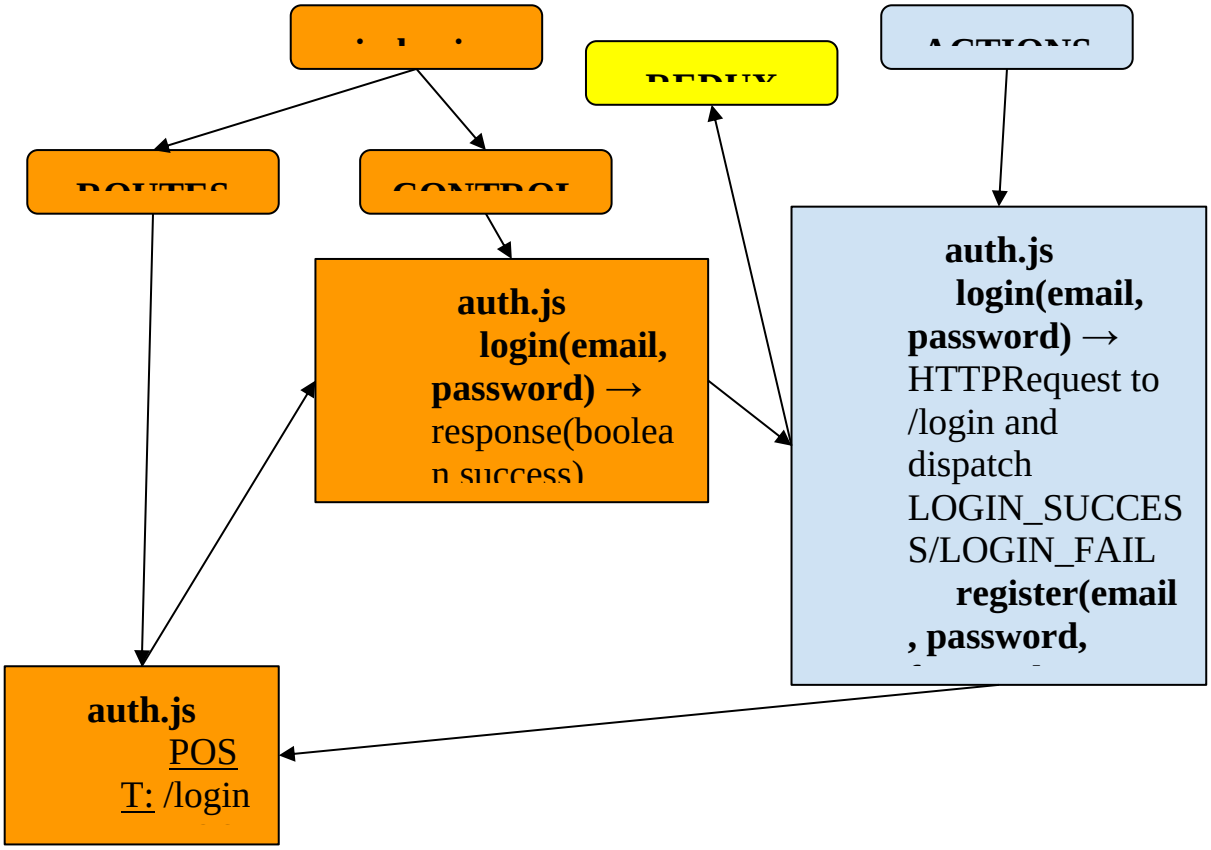
USER AUTHENTICATION



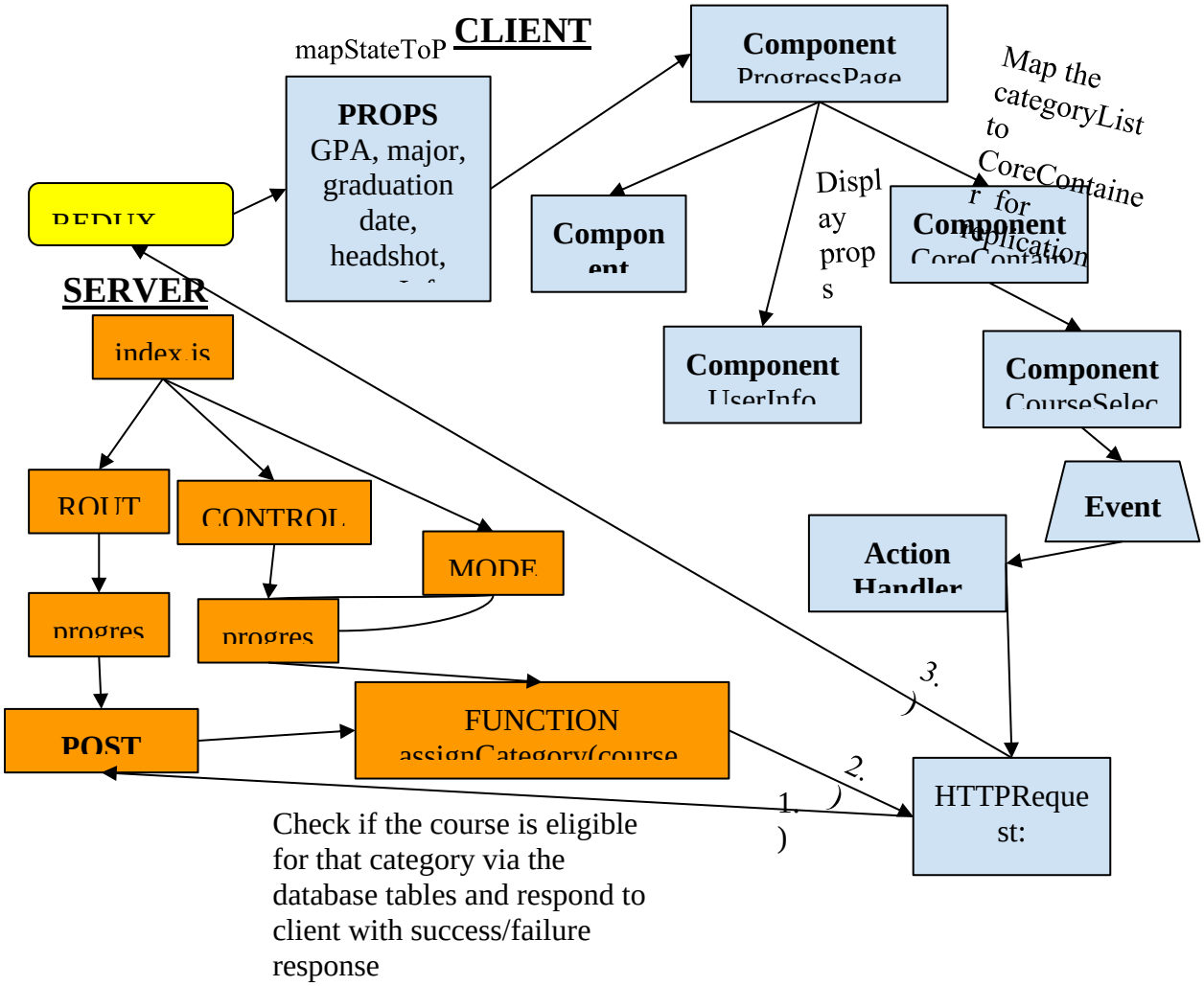
Above is the application trace for user authentication. Upon registration/login form submission, the application will execute the register or login action handlers, located in an **auth.js** microservice in our **actions** folder. The **register** function receives the data from the register form submission, including the first name, last name, email, password, major, headshot, and anticipated graduation date, and sends a POST request to the server via the **/register** route. On the other hand, the **login** function receives the data from the login form submission, including the email and password, and sends a POST request to the server via the **/login** route. The server has a routes folder containing microservices for each of the related endpoints of our application. The **auth.js** file of the routes folder contains the API endpoints for **/register** and **/login**. Upon reaching one of these endpoints, the application will call the respective controller function,

located within the respective microservice inside our **controllers** folder. Within the controller function, the server will make validation checks via the database and apply unit testing to ensure that the input data is sufficient. For instance, to register, the input must be of a reasonable length and format and a user with the same email must not already exist in the database. On the other hand, to login, the server will encrypt the password and compare it to the encrypted password within the user's entry in the database. For the register function, if the validation checks pass, then the server will create a new entry within our **Users** database table and encrypt the user's password. Finally, for both functions, the client will receive a response indicating whether the submission was successful. If successful, the client will receive the user data and a JSON web token and store the user information within the Redux store and the JSON web token within **localStorage** to handle sessions. After doing so, the client will redirect the user to the **Progress** page. If the submission fails, the view will display an error message by dispatching the LoginFail/RegFail action, updating the Redux store and re-rendering the component.

Here is the UML diagram describing the user authentication process:



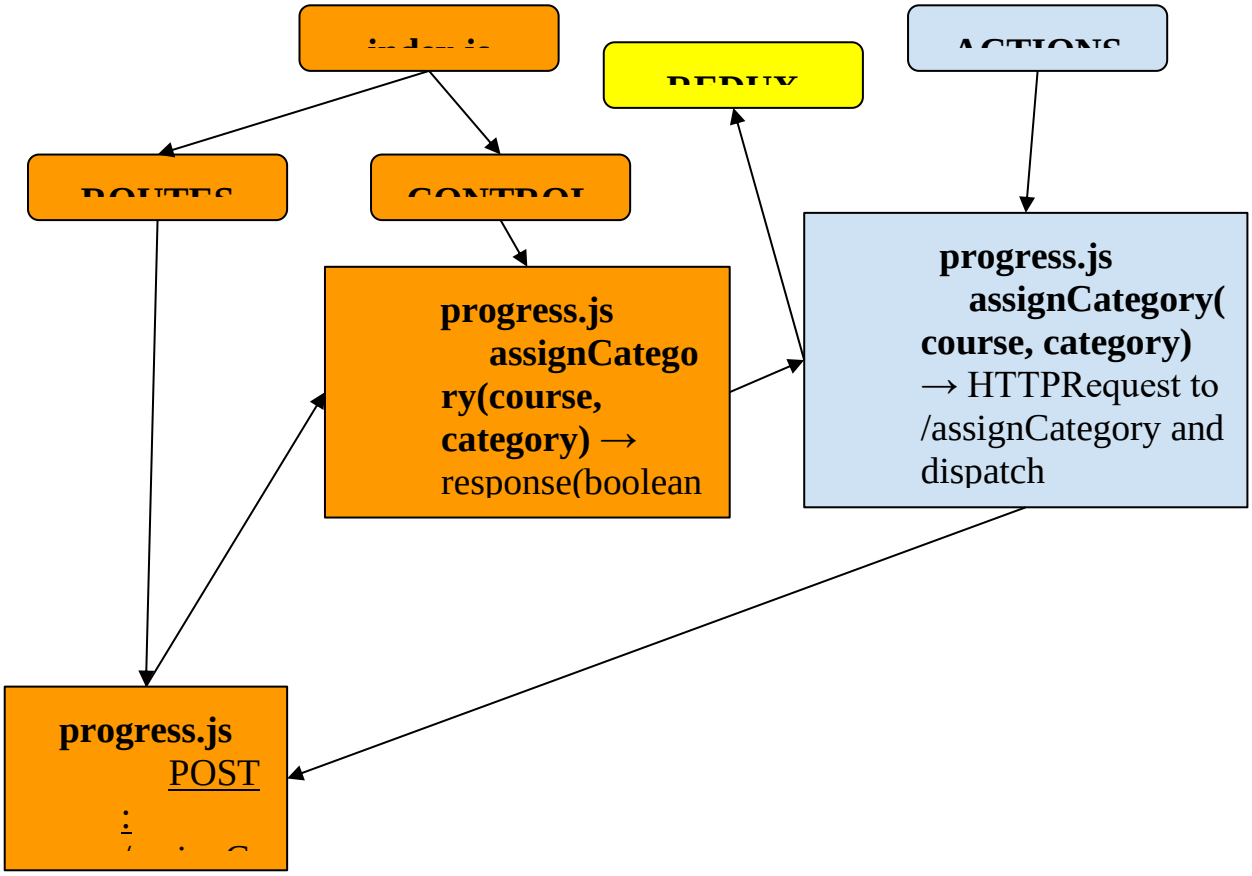
DEGREE PROGRESS



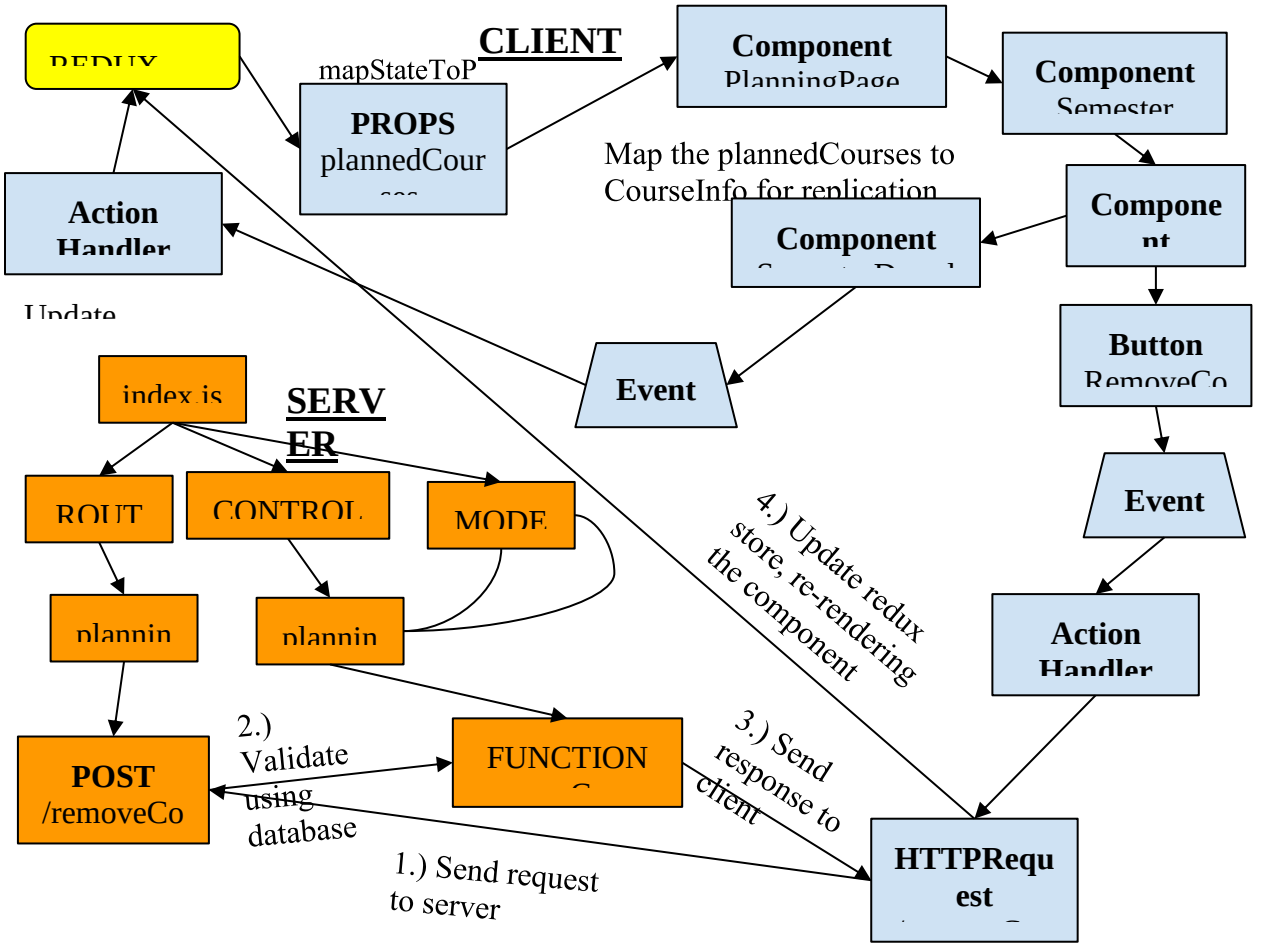
Above is the application trace for displaying the degree progress page and assigning a course to a core category. The progress page component will first receive the user information properties via the Redux store and display them through the UserInfo component. The **Core Container** components will act as placeholders for each of the different core requirements and each include a **CourseSelection** component for users to select an eligible course to fulfill that respective category. Upon selection, the client will call the assign category action handler, which will receive the selected course and respective category and send a HTTP post request to the

/assignCategory endpoint on the server. There is a **progress.js** route on the server dedicated to API endpoints for the progress page and a **progress.js** controller to handle the functionality of these API endpoints. The **assignCategory** function will receive the category and course and validate whether the course is eligible for that category via the database tables. A few of these unit tests include checking whether the course name is valid, if the user has actually taken the course, and if the course fulfills that category. If the validation tests pass, the server will update the respective database tables and send a successful response to the client, dispatching an action and updating the Redux store, re-rendering the **CoreContainer** component. Otherwise, the client will receive a response indicating that the assignment request is invalid and display an error message on the view. The default background color of the **CoreContainer** component will be red, indicating that the core requirement has not yet been fulfilled. However, if the validation tests pass, then the background color of this component will change to bright green to indicate that the user has successfully fulfilled the core requirement. Finally, the **progressBar** component will accumulate the number of credits a user has taken via a **coursesTaken** prop to indicate how close a user is to degree completion.

Here is the UML diagram for the degree progress page functionality:



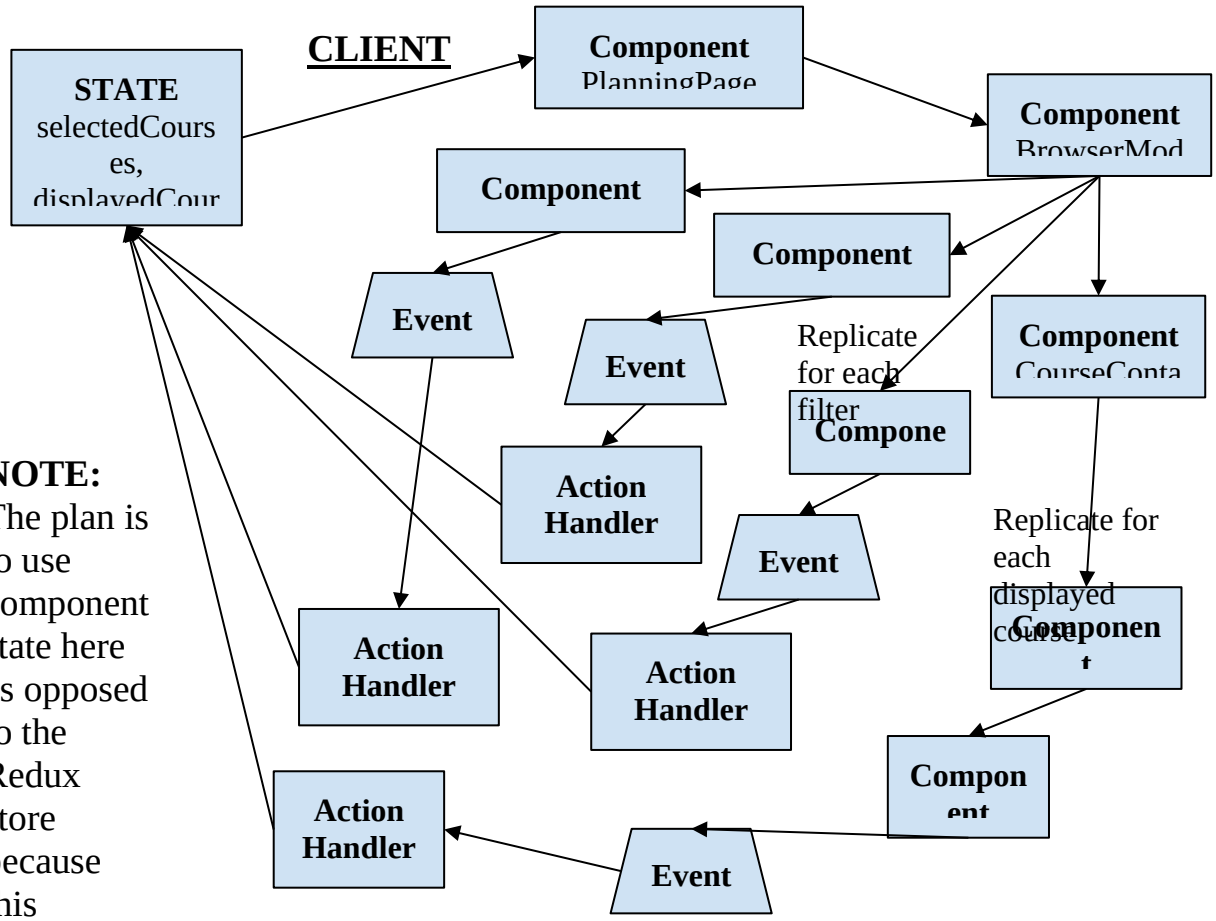
DEGREE PLANNER: SEMESTER COMPONENT



This is the application trace for the pieces of the degree planner that handle switching which semester a user is currently viewing and removing a course from a semester. The **PlanningPage** component retrieves the planned courses dictionary property from the user's data entry in the Redux store. For each key in this dictionary, the client replicates the **Semester** component. Then, for each course in a list under each key in the dictionary, the client replicates the **CourseInfo** component. This sub-component contains a button that triggers an action handler to remove the respective course located in the **planner.js** microservice in our **actions** folder.

This action handler sends an HTTP request to the API route /removeCourse on the server located in our **planning.js route** microservice. Upon receiving the respective course and semester in the request, the server passes this information to the removeCourse function located in our **planning.js controller** microservice. The controller then applies unit testing to validate that the semester key and course exist in the database tables. If these validation checks pass, the controller removes this entry from the database and responds to the client indicating that the removal was successful. The client then dispatches an action to update the **plannedCourses** property in the Redux store, causing the **Semester** component to re-render. If the tests fail, the controller's response indicates that the removal failed, causing the client to display an error message.

DEGREE PLANNER: BROWSER COMPONENT #1

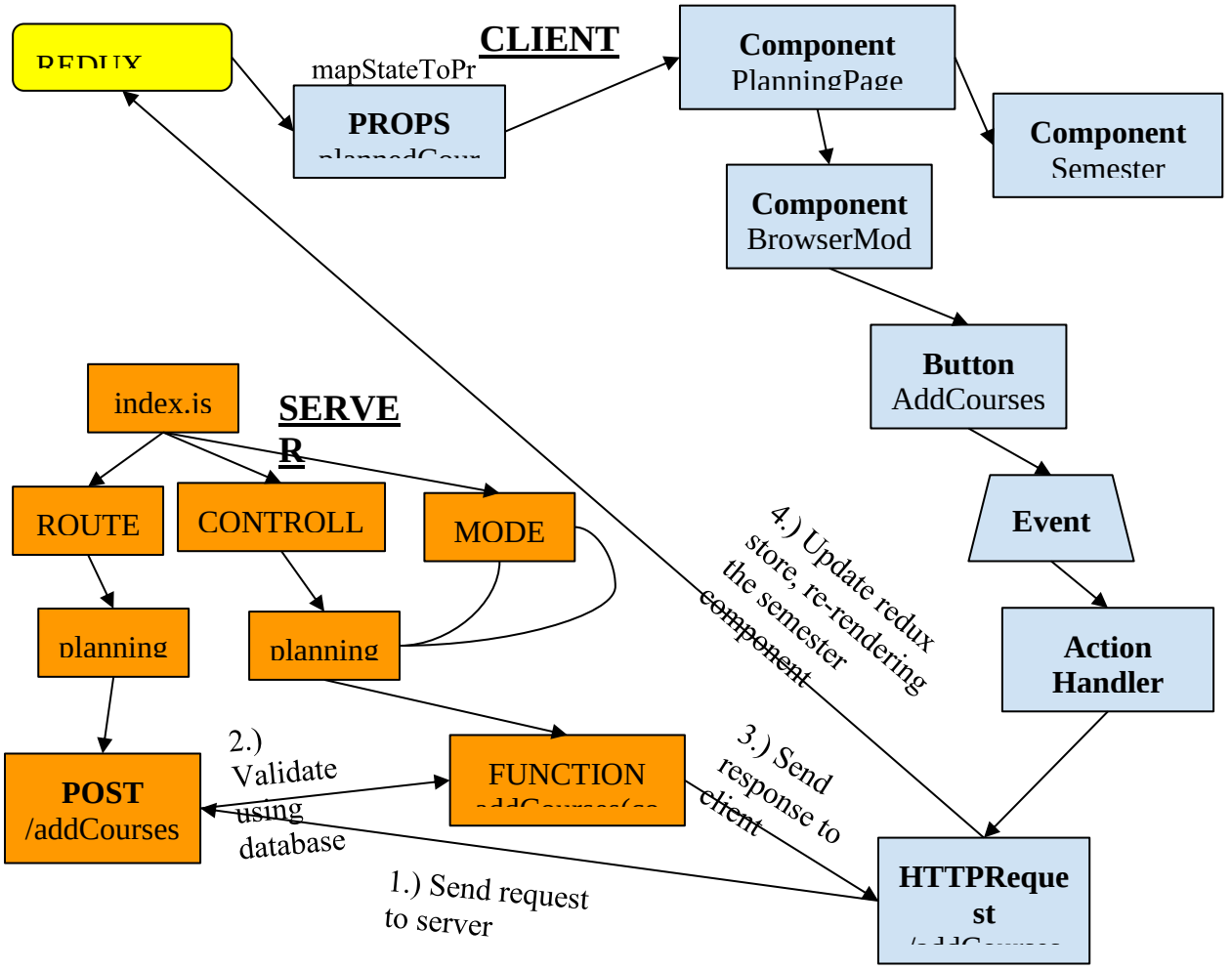


Within the planning page, there will also be a modal component that allows users to browse for courses to plan for the specified semester. The above figure models the aspects of the form that alter the **PlanningPage**'s state; we hope to avoid using the Redux store for these fields because they are irrelevant to other components. The layout of the browser modal includes the semester dropdown at the top, the course filters underneath, followed by the search bar, the

results container, and the submit button. The semester dropdown allows users to select from a range of eligible semesters from which to assign courses. Upon selecting a semester, the client executes the “select semester” action handler, located within the planning.js microservice in our **actions** folder. This updates the state of the PlanningPage component to reflect the chosen semester. As stated above, there will be a variety of course filters that allow users to reduce the amount of search results. Each of these filters will have a **SearchFilter** component that updates the **PlanningPage**’s state upon selection. The search bar component will allow users to browse for specific courses, and the PlanningPage’s state will update as the user is typing in the search bar. Finally, the course container component will display the results, given the enabled filters and the content within the search bar. For each course search result, there will be a **CourseInfo** sub-component that includes a check box component, allowing users to toggle the courses they wish to select. When a user selects a course, this will execute the “select course” action handler, also located within the planning.js actions microservice, updating the PlanningPage’s state and re-rendering the component. In addition, each time **appliedFilters** and **displayedCourses** aspects of the PlanningPage’s state change, the **CourseContainer** component will re-render, creating **CourseInfo** components that reflect the user’s choices.

The action handlers for this aspect of the browser functionality will operate within the scope of the PlanningPage component, removing the need for the Redux store for this aspect of the application. Therefore, we do not plan to modularize these action handlers as they will be calling the **setState** function to update the state of the PlanningPage component. For instance, within the **PlanningPage**’s state, we can have a **setCourseList(courseList)** function that updates the **selectedCourses** aspect of the state. This will require using the **useState** React hook. The below diagram will describe what happens upon submission of this course selection form.

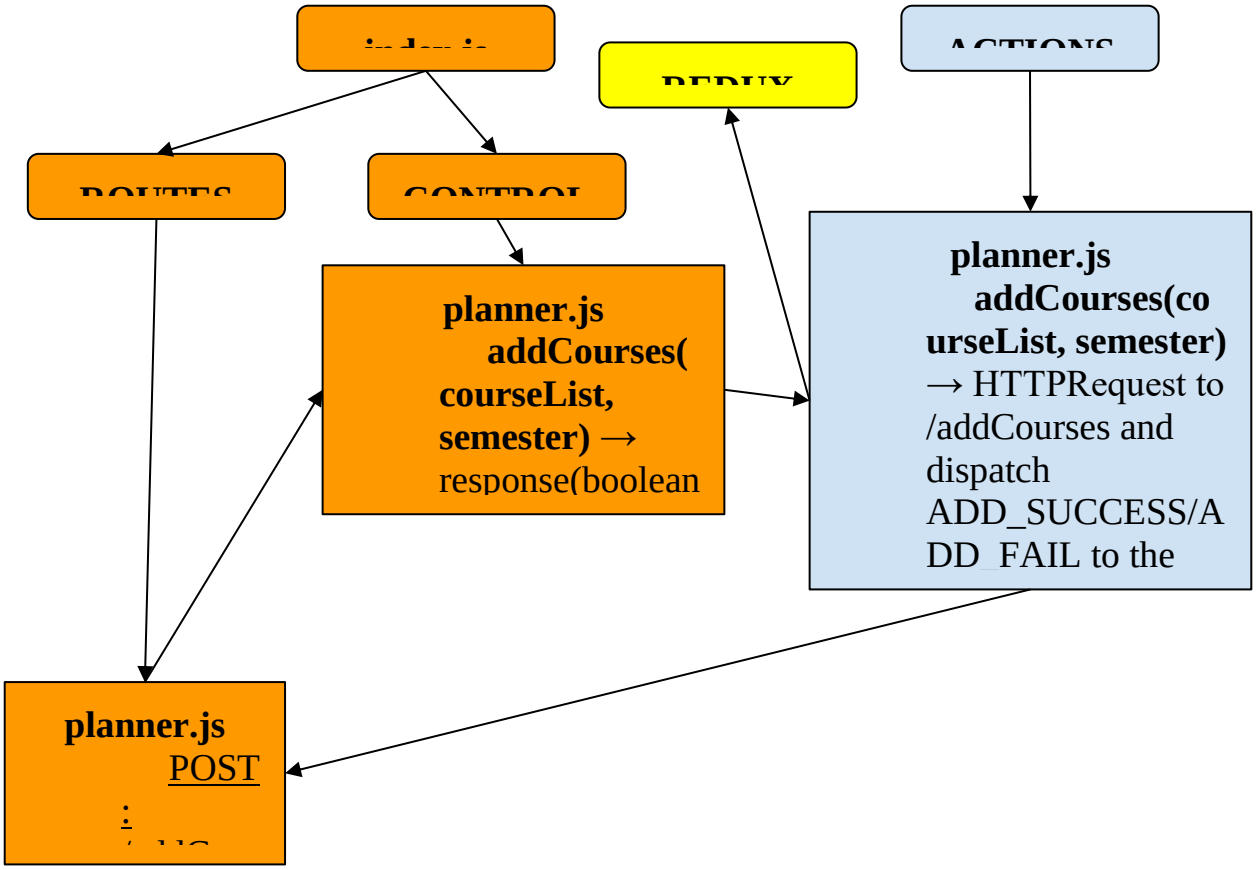
DEGREE PLANNER: BROWSER COMPONENT #2



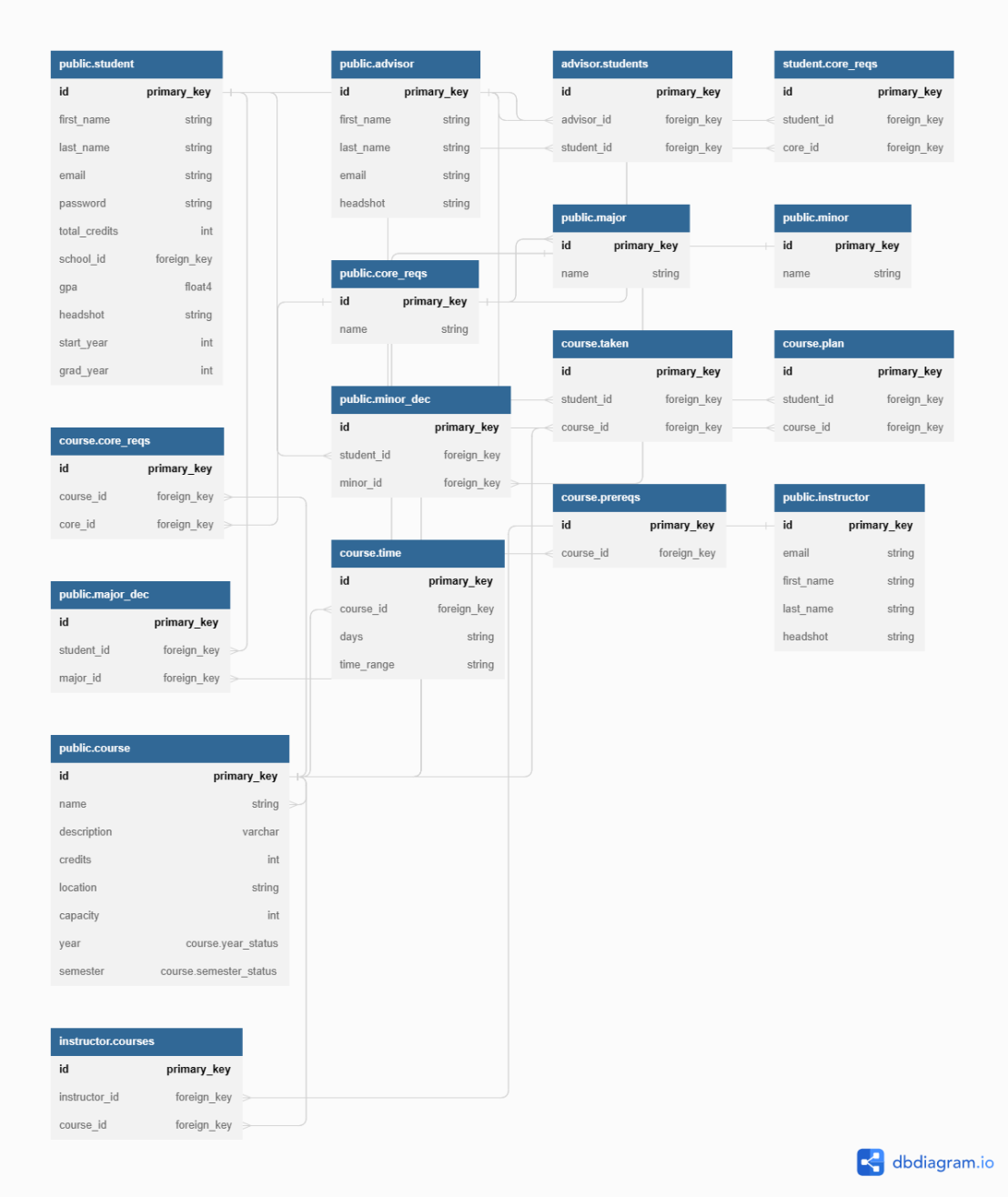
The final aspect of the browser functionality on the Planning page involves submitting the user's selected courses, updating the respective database tables on the server, and re-rendering the semester component to indicate the changes. Upon clicking the add courses submission button at the bottom of the browser form, the client will execute an action handler to add the courses, which will send an HTTP request to the server on the **/addCourses** API endpoint with the list of selected courses and the respective semester. This endpoint is located

within the **planning.js route** microservice in the **Routes** folder on the server. Upon receiving the request in this route, the server will execute the **addCourses** controller function, located within the **planning.js controller** microservice. This function will contain a series of unit tests, including checking if all the courses in the list exist, if the user is eligible to enroll in them based on courses taken or prior semester plans, and if the semester is valid based on the user's anticipated graduation date. If these validation tests pass, the server will update the database tables accordingly and send a response to the client indicating that the course assignment was successful. Upon receiving this response, the client will then dispatch an action the **Redux store** to update the **plannedCourses** in the state, causing the semester component to re-render. If the unit tests fail, the server will send a response to the client indicating that, and the client will display an error message.

Here is the UML diagram for the server-client functionality of the degree planner:



Finally, here is our database schema for this application. Note that the advisor and administration functionalities essentially exist as barebones for our aspirant scope should we have sufficient time to work on that. For that reason, we did not elaborate on those features within this design report. The other database tables directly relate to the requirements within our Minimum Viable Project Scope (MVP), many of which aim to normalize many fields of course information.


public.student
id primary_key
first_name string
last_name string
email string
password string
total_credits int
school_id foreign_key
gpa float4
headshot string
start_year int
grad_year int
public.advisor
id primary_key
first_name string
last_name string
email string
headshot string
advisor.students
id primary_key
advisor_id foreign_key
student_id foreign_key
student.core_reqs
id primary_key
student_id foreign_key
core_id foreign_key
public.major
id primary_key
name string
public.minor
id primary_key
name string
public.core_reqs
id primary_key
name string
course.taken
id primary_key
student_id foreign_key
course_id foreign_key
course.plan
id primary_key
student_id foreign_key
course_id foreign_key
public.minor_dec
id primary_key
student_id foreign_key
minor_id foreign_key
course.core_reqs
id primary_key
course_id foreign_key
core_id foreign_key
course.prereqs
id primary_key
course_id foreign_key
public.instructor
id primary_key
email string
first_name string
last_name string
headshot string
course.time
id primary_key
course_id foreign_key
days string
time_range string
public.major_dec
id primary_key
student_id foreign_key
major_id foreign_key
public.course
id primary_key
name string
description varchar
credits int
location string
capacity int
year course.year_status
semester course.semester_status
instructor.courses
id primary_key
instructor_id foreign_key
course_id foreign_key
dbdiagram.io