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

Dr. Mongan

CS-375: Software Engineering

26 March 2023

Academic Planner Test Plan Report

Our academic planner application uses a development stack comprising of React and Redux to handle client functionality on the front end and Node and Express to create our microservices on the back end. In addition, we write to and retrieve data from a SQLite3 database on the back end. With the React framework, we can divide our view into separate entities called components. Each component has their own respective JavaScript and CSS modules, detailing their functionality and styling respectively. For instance, a few of our components such as our header include forms, allowing users to submit data and information to the server. Each API endpoint on the server directs the client request to its respective controller function, within a server-side microservice module designed for the respective component. With all the actions that users can perform, our goal is to ensure that users have a smooth experience and do not run into errors due to faulty inputs. Therefore, we have listed a series of white box and black box tests for each of our functions.

Our header component includes a button that opens a form where users can toggle between signing in and registering. To handle this functionality, we are using React state, enabling us to easily store information within our components pertaining to simple client-side actions that do not involve the server. In this case, for each of our buttons that open a separate form (or modal, as described in Bootstrap documents), we simply need to store the state of a toggle variable

within our component states. Our application also includes a module that comprises of action handlers, facilitating requests to the server-side of our application. For instance, when a user submits the registration form, the client sends the inputted email, password, first name, last name, anticipated graduation date, major, and headshot to the respective action handler, which then sends a request to the server on the "/register" API endpoint with this data. Before updating the database, however, we want to validate that these parameters meet the correct specifications, preventing any issues from occurring. Here are the required formats for each of these variables:

- **register(email, password, fName, lName, gradDate, major, headshot) → json(message: string)**
  - **Email:** Cannot be an empty string and must follow this regular expression: **/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,15}$/**
    - ****We must check that this email is unique (there is no existing account in the database including this email).
    - A few examples would be john.doe@example.com and jane_doe@example.co.uk. This is the standard regular expression for email addresses.
  - **Password:** Cannot be an empty string and must follow this regular expression: **/^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/**
    - **Note that we apply encryption by serializing passwords through the "bcrypt" library to keep them secure
    - A few example inputs would be "Abc123", "P@ssword123", "Qwerty12", "Passw0rd", and "1Abcdefg", etc.

- **fName:** Only composed of alphabetical letters and one hyphen (if they have a two-word name such as Mary Kate, put a hyphen)
    - The first letter of the first name must be upper case. If there is a hyphen in the first name, the first character after the hyphen must be uppercase.
- **lName:** Only composed of alphabetical letters and one hyphen (if they have a two-word name such as Mary Kate, put a hyphen. Ensure that the first letter after the hyphen is uppercase). The first letter of the last name must be capitalized.
    - A few examples of input would be “Veca-Schilling”, “Gillette”, etc.
- **gradDate:** There are available options for graduation date listed within a separate JSON module that is shared between the client and the server. The server checks this input against the array of semester keys (appears as a dropdown menu on the client regardless). A semester key begins with the first letter of the respective season followed by a four-digit year, such as “S2024.”
    - A few examples of this format would be “S2022”, “F2021”, etc.
- **Major:** Similarly, there is a separate JSON module for majors that is shared between the client and the server. We use a dropdown menu for major selection on the client, but the server does an additional check to ensure that the major is within the “majors array” in this module.
    - A few examples of input would be “Computer Science”, “Mathematics”, etc.
- **Headshot:** The server validates that the user’s uploaded headshot includes any of the following file types: **PNG, JPEG, JPG**.
    - A few examples of input would be “example.png”, “test.jpg”, etc.

If all these parameter validation tests pass, then we proceed to insert a new row in the **StudentInfo** table of the database that includes all this information. Regardless of whether the registration attempt was successful, the client receives a response from the server including feedback related to the attempt. For instance, if there is already an existing account with the requested email address, the view displays: “An account with this email address already exists!” To display this, we include the message parameter of the server’s response within the Redux store and access it from within the Header component. Essentially, we apply a black box test by using “conditional rendering” (displaying an HTML element based on a condition). To elaborate, if the message from the server exists (is not null), we render an Alert component, displaying this feedback on the view.

Following a successful account registration process, a user can instantly sign into their newly created account. The sign-in functionality follows a similar architecture, using different functions within the same action handler and controller modules. Upon submitting the sign-in form, the “login action handler” sends a request to the “/login” API endpoint on the server containing the user’s email and password input. Following this, the respective controller function on the server both validates these parameters and runs a few database checks to validate that the user can log in. Here are the following white box tests for our sign-in controller function:

- **login(email, password) → json(message: string, accessToken: string)**
  - **Email:** cannot be an empty string and must follow this regular expression: /^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/
    - A few examples would be john.doe@example.com and jane_doe@example.co.uk. This is the standard regular expression for email addresses.

    - **There is a database check to ensure that the email exists within a row of the StudentsInfo table.
- **Password:** cannot be an empty string and must follow this regular expression: /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,15}$/
    - This regular expression asserts that the password input is between six and fifteen characters in length, contains at least one capital letter, one digit, and one lowercase letter.
    - A few example inputs would be “Abc123”, “P@ssword123”, “Qwerty12”, “Passw0rd”, and “1Abcdefg”, etc.
    - **We again use the “bcrypt” library to serialize the user’s password input and compare it to the serialized password stored in the user’s respective row of the database table

If all these white box tests pass, then the client receives a response from the server containing an access token, which the server uses to validate that clients are authenticated and can visit protected routes. Regardless of whether the validation tests pass, the client receives a feedback message from the server and follows the same process described above to display it on the view. If the sign-in process was successful, however, we do not display any feedback message on the view. For our black box testing for the sign-in process, we check whether the access token exists (indicating whether the process was successful or not) and store it within local storage on the client (so that we can store it in the header of subsequent HTTP requests). In addition, we check whether there is a feedback message and display it if so (in this case, there will never be both an access token and a feedback message).

The progress page follows the same development architecture as the user authentication system. To reiterate from previous reports, the progress page allows users to assign courses to core requirements. Therefore, upon loading the progress page, the client makes a request to the server to fetch the courses that a user has already assigned to core requirements. There is a table component within the progress page that includes cells for each of the core requirements, and we apply a color code to the fill color of the cells to indicate whether the user has fulfilled the corresponding core requirement. To elaborate, a green fill indicates that the user has fulfilled the requirement, whereas a red fill indicates that the requirement has not been fulfilled. Upon clicking any of the cells, we display a pop-up including the ID of the course that the user assigned to the corresponding core requirement of that cell. Unlike the current system, this core requirements table is more visually appealing and not complex.

To handle assigning courses to core requirements, we include a button at the top of this page that toggles a course assignment form. From there, users can select a core requirement from a dropdown menu (like the majors list in the registration form, we have a shared JSON file including a list of core requirements) and choose from a list of courses within their degree plan that fulfill the respective core requirement. Thus, to implement this system, we will also have to fetch the user's degree plan from the database as well. Upon submitting the form, we will pass the selected course and core requirement to the respective function within the progress page action handler module and send a request to the "/assignCategory" API endpoint, calling the respective function within the progress page controller on the server. Here are our intended white box tests for this function:

- **assignCore(courseId, coreId) → json(message: string)**
    - **Course:** The course ID must not be an empty string or null.

    - We must check if the requested Course ID exists within the user's degree plan stored in the database.
    - We must check the course information database table to see if the course exists and fulfills the requested core requirement.
    - We must determine if the course can fulfill this core based on if the user has selected this course to fulfill another core requirement. We must consider certain conditions involving groups of core requirements (i.e. A, H) where a single course can fulfill more than one category.
- **Core:** The core must not be an empty string or null.
    - The requested core requirement must be included in the list of core requirements in the external JSON file described above.

If the course and core category inputs are valid as defined by the conditions above, then the database updates and the response to the client includes a message indicating that the user has successfully assigned the course to the respective category. Otherwise, the client receives a message indicating that this process has failed. For black box testing, we apply conditional rendering by following the same steps as described above to display the feedback message on the view.

For major requirements, we plan to implement a similar table system as the core requirements table, but we have not yet worked out the full details of this because each major has unique requirements. This feature may be a part of our aspirant scope.

The last component of the progress page is our implementation of a progress bar that indicates how many credits a student needs to graduate compared to how many they have currently planned. To handle this, we plan to send a request to compute the number of credits a

user has planned from their course plan stored on the client and then update the progress bar to represent such. The only thing we would have to test with this is that the computed credit amount is greater than zero. We would not have to check an upper bound for this computed credit amount because students can overload or receive credits from other institutions.

The final page of our application is the degree planner page, allowing users to add and remove courses to and from their degree plan. We follow the same action-controller system as outlined above, but this aspect of the application includes more functions that involve only component state as opposed to the back end. The page displays the course plan for a single semester and allows users to switch semesters, updating component state when a user selects a new semester on the dropdown menu. As described in the design report, we have a Semester component, which is essentially a container that includes CourseInfo components, smaller entities each including information about a course in the semester's plan. Upon navigating to this page, the client will initially send a request to the server on the "/fetchPlan" API route to obtain the user's degree plan stored within the database. If the user has no degree plan, the Semester component will not update. Otherwise, the client will replicate CourseInfo components for each of the courses listed in the respective semester's plan, validating that each of the courses has properly defined fields (the CourseInfo component for the respective course will not render otherwise). Each of the CourseInfo components contain a remove button in the top right corner, allowing users to remove the course from the plan. Upon clicking this button, the client passes the course ID and the semester key to the remove function within the planner action handler, sending a request to the server on the "/removeCourse" API endpoint, executing the respective function within the planner controller module on the server. Here are our white box tests for removing courses from the degree plan:

- **removeCourse(course, semester) → json(updatedPlan: [], message: string)**
    - **Course:** The course ID cannot be null or an empty string.
        - We must validate that a course exists in the database with this course ID with a database query on the CourseInfo table.
    - **Semester:** The semester key must start with either an “S” or an “F”, indicating the spring or fall semester respectively. The rest of the key must be a four-digit year. An example of this would be “S2024.”
        - We must also verify that the key makes sense for the user by comparing it to the user’s anticipated graduation date in the StudentsInfo table of the database (so we must write a query to do this). For instance, “S2025” would not make sense if a user’s anticipated graduation date were “S2024.”

If the white box tests pass, we run a database query to remove the course from the user’s degree plan in the database. We then send the updated plan to the client to update the Semester component’s view. For our black box tests, we once again apply conditional rendering, displaying an error message if the updated plan is an empty array or null. If the black box tests pass, we display the updated plan on the view. Regardless of whether the validation tests pass, the client receives a response from the server including a feedback message indicating whether the removal process was successful. We display this message using the same process outlined for the previous components. Our black box test for the feedback message includes verifying that the message is not an empty string or null and then conditionally rendering it based on that.

Arguably, the most important aspect of the planner page is the ability to add courses to the course plan. For this, we have a button located on the top right corner of the page that toggles a modal component for course browsing. We use component state to determine whether this modal

is currently open. As described in the design report, we have a series of filters and a search bar, allowing users to filter the full course list to facilitate the course selection process. In addition, we have a dropdown menu that allows users to select the semester in which they want to add the courses. Upon enabling any filters, choosing a new semester from the dropdown menu, or inputting a search query, we update component state to determine the updated course list that fulfills all these parameters and display it in the search results container. We apply some white box testing here by ensuring that the search query is not an empty string or null. Otherwise, we do not have to perform a lot of testing with our functions included within the semester component's state because React automatically handles that for us. Here is the list of functions we can execute from within the semester component's state:

- Browser Component:
    - toggleFilter(key) → void **(updates appliedFilters in component state)**
    - updateDisplay() → void **(checks appliedFilters and searchQuery in component state and filters the complete list of courses to update the displayedCourses variable in component state)**
    - selectCourse(key) → void **(updates selectedCourses in component state)**
    - selectSemester(key) → void **(updates selectedSemester in component state)**

Upon submitting the add courses form, we pass the course ID list and the semester key to the respective function within the planner action handler, sending an HTTP request to the "/addCourses" endpoint, calling the respective function within the planner controller module. Here are our white box tests for that controller function:

- **addCourses(courseList, semester) → json(updatedPlan: [], message: string)**
    - **courseList:** The course list must not be null or an empty array.

- We must validate that each of the course IDs exist within the database by running a select query on the CourseInfo table.
- We must also validate that the user has not already added any of the requested courses to any of their semester plans (uniqueness test; no duplicates throughout the whole plan).
- Finally, we must verify that the course is offered in the specified semester and year by querying the CourseInfo table. To handle all these tests, we can use SELECT database queries.
- A few examples of input would be: "[]", "['CS-174']", "['MATH-111', 'CS-173']."

- **semester:** This must be a valid semester key with the same format as described above (first character indicates a season, and the next four digits indicate the year).
  - A few examples of input would be "S2023", "F2022", etc.
  - Like with the tests for the remove course function, we must also verify that the key makes sense for the user by comparing it to their anticipated graduation date stored in their respective entry of the StudentsInfo table.

If the course ID list and the semester key are both valid based on these conditions, we update the database tables and send the full updated plan to the client along with a feedback message indicating that this process was successful. We can then verify that the output is valid by using conditional rendering to not display the semester component if the updated plan does not exist. We follow the same steps as outlined above for displaying the feedback message on the view.

All the above tests for each page of our application cover all the unique features clients can use to effectively plan out their degree. Most of the tests validate that the parameters match their specific formats and make logical sense for the individual user. To handle the latter case, we plan to run database queries to compare fields such as anticipated graduation date, email, and password with the user inputs to determine whether to update the database tables. Therefore, in the different sections above, we account for all the different cases involving user inputs and actions, demonstrating sufficient code coverage for each decision branch. Breaking up our entire application into JavaScript modules aids in our implementation of validation tests because we can isolate and cover all the code within all the various aspects of our application.

We have additionally developed a script for user acceptance testing to ensure that clients can easily understand how to access and use the features of our application:

Test Number	Task	Expected Output	Requirement ID
Precondition: Login Form	Click the login button on the right side of the header (located at the top of the page) and be directed to the login form popup.	Ensure that the login form appears with email and password input fields.	**1.1a:** Login Form
Login Success	Enter your respective email and password and press the sign-in button located at the bottom right of the screen.	Gain access to the protected routes of the application: the degree planner and my progress page. The login button should now update to say “logout”	**1.1a:** Login Form
Login Fail	Follow the same steps as “login success” but enter an invalid email or password on the login form.	**Receive any of these feedback messages:** “User not found”, “Invalid password”, “Database not initialized”	**1.1a:** Login Form

Test Number	Task	Expected Output	Requirement ID
Logout	Press the logout button located to the right of the header.	Verify that you are redirected to the home page and can no longer access the “my progress” and “degree planner” pages.	**1.2:** Token System
Precondition: Registration Form	If not registered, click the “login button” on the right side of the header and click the “create account” button in the bottom right corner of the registration form.	View the form for user registration with the “first name”, “last name”, “anticipated graduation date”, “major”, and “headshot” fields.	**1.1a:** Registration Form
Email/Password	In the email slot, please enter a valid email. In the password slot, please enter a password at least eight characters long with at least one capital letter.	Receive an alert if there are invalid formats in your email and/or password input	**1.3:** Register Validation
Anticipated Graduation Date	Select a graduation year from the respective dropdown menu.	The dropdown updates to indicate your selection.	**1.3:** Register Validation
Major	Select a major from the dropdown menu	The dropdown updates to indicate your selection.	**1.3:** Register Validation
Headshot	Upload a photo of your headshot in either a PNG, JPG, or JPEG format.	Ensure the form displays the file name. Receive an alert if there is an invalid file format.	**1.3:** Register Validation
Create Account	After all selections are made, press the register button located in the bottom right of the screen.	Receive a feedback message indicating whether your registration process is successful. If so, you will now be able to return to the sign-in form and input your new	**1.3:** Register Validation

Test Number	Task	Expected Output	Requirement ID
		credentials. If not, you will receive an error message indicating the problem, such as, "Account with that email already exists", "Database not initialized", or "Invalid <field> format."	
Navigation	Access the different pages of the application from the header located at the very top of the page.	**Precondition**: Sign in and gain access to the course planner and my progress pages. If not logged in, be redirected to the home page when attempting to access those pages.	**1.4:** React Router
1.1 Navigation	Click the graduation cap icon	Access the "my progress" page if logged in.	**1.4:** Progress Navigation
1.2 Navigation	Click the home icon	Redirection to the landing page of the application.	**1.4:** Home Navigation
1.3 Navigation	Click the calendar icon	Access the "degree planner" page if logged in.	**1.4:** Planner Navigation
Precondition: Degree Planner	Complete Test #1.3 first.	Verify that you see a label at the top left of the page indicating your first eligible semester along with its respective course plan in the center of the page. Ensure that you see an "Add Courses" button and "Switch Semester" dropdown on the right of the page.	**3.2a:** Fetch Plans
2. View a different semester's plan	Access the dropdown menu located in the top right corner of the page and select from	You should see your respective plan for the chosen term or an error message indicating that	**3.2c:** Semester Plan Dropdown

Test Number	Task	Expected Output	Requirement ID
	a list of eligible semesters.	there was an issue accessing the information for this term.	
Precondition: Course Browser Modal	Navigate to the degree planner. Click the Course Browser icon located in the top right corner of the screen	View the course browser popup form. Verify that the first fifty courses of the course catalog appear in the search results box at the bottom of the form.	**3.1:** Course Browser Form
3. Course Filters	Toggle the various “core requirement” and “semester offered” filters	Verify that the course browser results updates so that all the courses that match those filters appear.	**3.1c:** Course Filters
4. Search Course	Search by course ID or name and press the magnifying glass button to the left of the text box.	Ensure that the respective courses that fit your search query appear.	**3.1a:** Course Search Bar
5. Switch Semester	From the switch semester dropdown menu, select the semester in which you wish to add the courses	The dropdown should update to indicate the new selection	**3.1d:** Course Selection Semester Dropdown
6. Add Courses	Press the add courses button located at the bottom right of the form.	Follow test #2 and ensure that the respective semester’s plan has updated.	**3.1e:** Browser Form Submission Button **3.3:** API Endpoint
7. Remove Course	Follow test #2 to navigate to the semester’s plan where the respective course is located. Press the remove button located in the top right of the course’s information container.	The respective semester’s plan should update so that you no longer see the container for the respective course.	**3.2b:** Course Removal Button **3.3:** API Endpoint

Test Number	Task	Expected Output	Requirement ID
Precondition: My Progress Page	Follow test #1.1 to navigate to the “my progress” page.	Verify that there is a core requirements table located in the center of this page.	**2.1:** Fetch Assignments
8. Red Requirements	Click on cells in the core requirements table that are filled red.	Verify that a form pops up with a core requirement dropdown menu and a list of eligible courses from your plan that fulfill that requirement. Ensure that the default value of the dropdown menu is the name of the respective core requirement that you pressed.	**2.2a:** Core Fulfillment
9. Green Requirements	Click on cells in the core requirements table that are filled green.	A popup notification should display on the screen indicating which course fulfills this requirement.	**2.2a:** Core Fulfillment
10. Requirement Assignment Form	Click the requirement assignment form toggle button located in the top right corner of the page.	Verify that you have the same expected output as test #8, with no default value of the core requirement dropdown menu.	**2.2c/2.4:** Core Assignments
11. Credits Progress Bar	At the top left corner of the screen, there should be a progress bar indicating how many credits you have mapped out compared to the minimum number needed to graduate (128).	Verify that the computed number of credits is consisted with your degree plan and the proportion of the computed number out of 128 looks correct visually on the progress bar.	**2.3:** Progress Bar