Front End Testing Implementation
To test my front-end software, I decided to install vue-jest, which is the jest package that supports vue.js applications. In order to get my testing setup working, I also needed to install babel-jest and various babel packages, which assist in transforming javascript and vue files into the type of javascript that jest knows how to interpret.

Front-End Test Cases
When it comes to my front-end code, I have a total of 8 vue components / vue files. To keep things simple and easy to understand from a developers standpoint, I decided to create one test file (.spec.js file) for each component. Furthermore, within each test file, the overall test case that is being run is named exactly as the name of the component. Thus, in the coverage report and the testing consoles, it will be easy to understand which test cases fall under which component. In the following sections, I will summarize each test file, and which test cases are taken into account.

Login.spec.js
This test file tests the front-end code for Login.vue. This test file starts by testing some of the branches within the actual HTML section of the file, to ensure that the UI is responsive and acts appropriately. Next within this file, I test the anchor tags for Google and LinkedIn buttons, to ensure that when the user clicks on these, they are redirected to the appropriate page. After that part of the code is tested, my test cases cover the login functionality for manual login. These test cases run through each individual case: errors related to invalid user inputted data, errors related to backend failures, errors related to callback functionality code, and finally the successful login case.

Home.spec.js
This test file tests the front-end code for Home.vue. My Home.vue file essentially comes down to testing if my components work together. There are many functions within this file that open and close modals, and event callbacks from other components. For each function, I have a test case that ensures the correct functions are being called. Additionally, there are test cases for each emitted event from a child component, to ensure the correct callback function is called.

NavBar.spec.js
This test file tests the front-end code for NavBar.vue. This test file is very small, and essentially just tests if the navigation bar on the left side of the home screen redirects to the correct route on click. There is a test case for each navigation item.

Card.spec.js
This test file tests the front-end code for Card.vue. These test cases are mainly user interface related. There are branches in my HTML code to only render items if the data is valid / exists. The test cases run through each one of these branches. Additionally, there are two test cases for the update and delete icons to ensure that they emit the correct events to the Home.vue file.

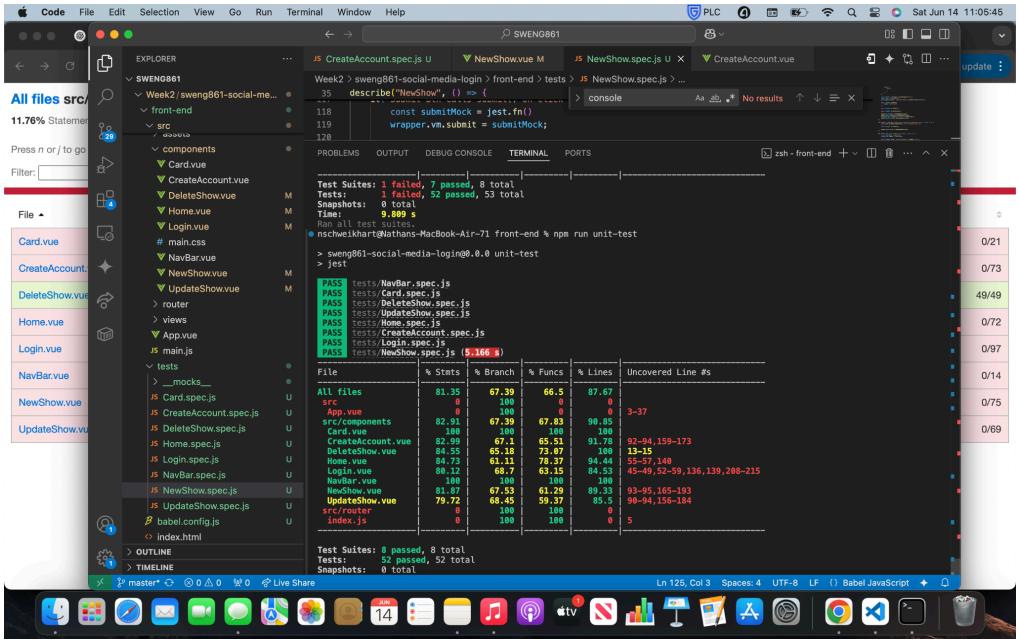
NewShow.spec.js
This test file tests the front-end code for NewShow.vue. This file essentially tests the front-end handling of the createShow API call. There is one test case for each edge case related to this API call: errors related to missing or invalid data, errors related to backend failures, errors related to callback functionality, and the success edge case.

UpdateShow.spec.js
This test file tests the front-end code for UpdateShow.vue. This file essentially tests the front-end handling of the updateShow API call. There is one test case for each edge case related to this API call: errors related to backend failures, errors related to callback functionality, and the success edge case.

DeleteShow.spec.js
This test file tests the front-end code for DeleteShow.vue. This file essentially tests the front-end handling of the deleteShow API call. There is one test case for each edge case related to this API call: errors related to backend failures, errors related to callback functionality, and the success edge case. Additionally, there are a couple test cases related to branches in the HTML / UI.

CreateAccount.spec.js
This test file tests the front-end code for CreateAccount.vue. This file essentially tests the front-end handling of the createAccount API Call. There is one test case for each edge case related to this API call: errors related to missing or invalid data, errors related to backend failures, errors related to callback functionality, and the success edge case.

Coverage(s)
In Vue-Jest, the package automatically generates a coverage report that splits it into % Statements Covered, % Branches Covered, % Functions Covered, and % Lines Covered. It will furthermore highlight the file in red if you achieved less than 50% coverage, yellow if you have between 50% and 80% coverage, and green if you have 80% or more coverage. Below is a screenshot of the front-end coverages for my test cases…



Challenges Faced in Front-End Testing
Installing vue-jest is a massive challenge. There are only one or two exact combinations of the versions for each package that work together, and installing the current versions of each package does not work. Figuring this out was difficult, but once it was working it was easy to use. One other challenge faced was testing my UpdateShow.vue file. The Coverage Report on this indicates that the coverage level is between 50% and 80% (highlighted in yellow), but looking further into the coverage report itself, it is reporting back false data. It is indicating things like CSS styles, and HTML label tags are statements and branches that are not covered. Additionally it is saying function statements are not coverage when I know for a fact they are and I validated that it's covered in the test cases itself and via console logs to ensure the code reaches that point.

Back-End Testing Implementation
For my backend software testing, I decided to install jest. Since my back end route files are already in native JavaScript that Jest can interpret, there was no need to install babel or other packages to support it.

Back-End Test Cases

When it comes to my back-end code, I have a total of 4 route files. To keep things simple and easy to understand from a developers standpoint, I decided to create one test file (.test.js file) for each route. Furthermore, within each test file, I declare a new set of tests for each endpoint within that file. Thus, in the coverage report and testing console, it is easy to understand which tests relate to which endpoint.

google.test.js

This test file tests the API endpoints within my google.js route. The test cases here cover the /auth/google and /googleCallback endpoints. There is one test case for /auth/google, which ensures that it redirects to the google authentication page. There is one test case for each of the edge cases for /googleCallback: errors related to google authentication, database functionality if the user exists / does not exist, and redirection.

linkedin.test.js

This test file tests the API endpoints within my linkedin.js route. The test cases here cover the /auth/linkedin and /linkedinCallback endpoints. There is one test case for /auth/linkedin, which ensures that it redirects to the linkedin authentication page. There is one test case for each of the edge cases for /linkedinCallback: errors related to linkedin authentication, database functionality if the user exists / does not exist, and redirection.

shows.test.js

This test file tests the API endpoints within my shows.js route. This test file has each of the CRUD operations test cases. For each API endpoint here, there is one test case for each of its edge cases: errors related to missing data, errors related to invalid data, errors related to database failures, errors related to existing or non-existing shows, and the success edge cases.

users.test.js

This test file tests the API endpoints within my users.js route. This test file tests the /createAccount and /login API endpoints. For each API endpoint here, there is one test case for each of its edge cases: errors related to missing data, errors related to if the user exists or not, errors related to password hash comparison, and the success edge cases.

Coverage(s)

In Jest, the package automatically generates a coverage report that splits it into % Statements Covered, % Branches Covered, % Functions Covered, and % Lines Covered. It will furthermore highlight the file in red if you achieved less than 50% coverage, yellow if you have between 50% and 80% coverage, and green if you have 80% or more coverage. Below is a screenshot of the front-end coverages for my test cases…