Showcase Project

Introduction:

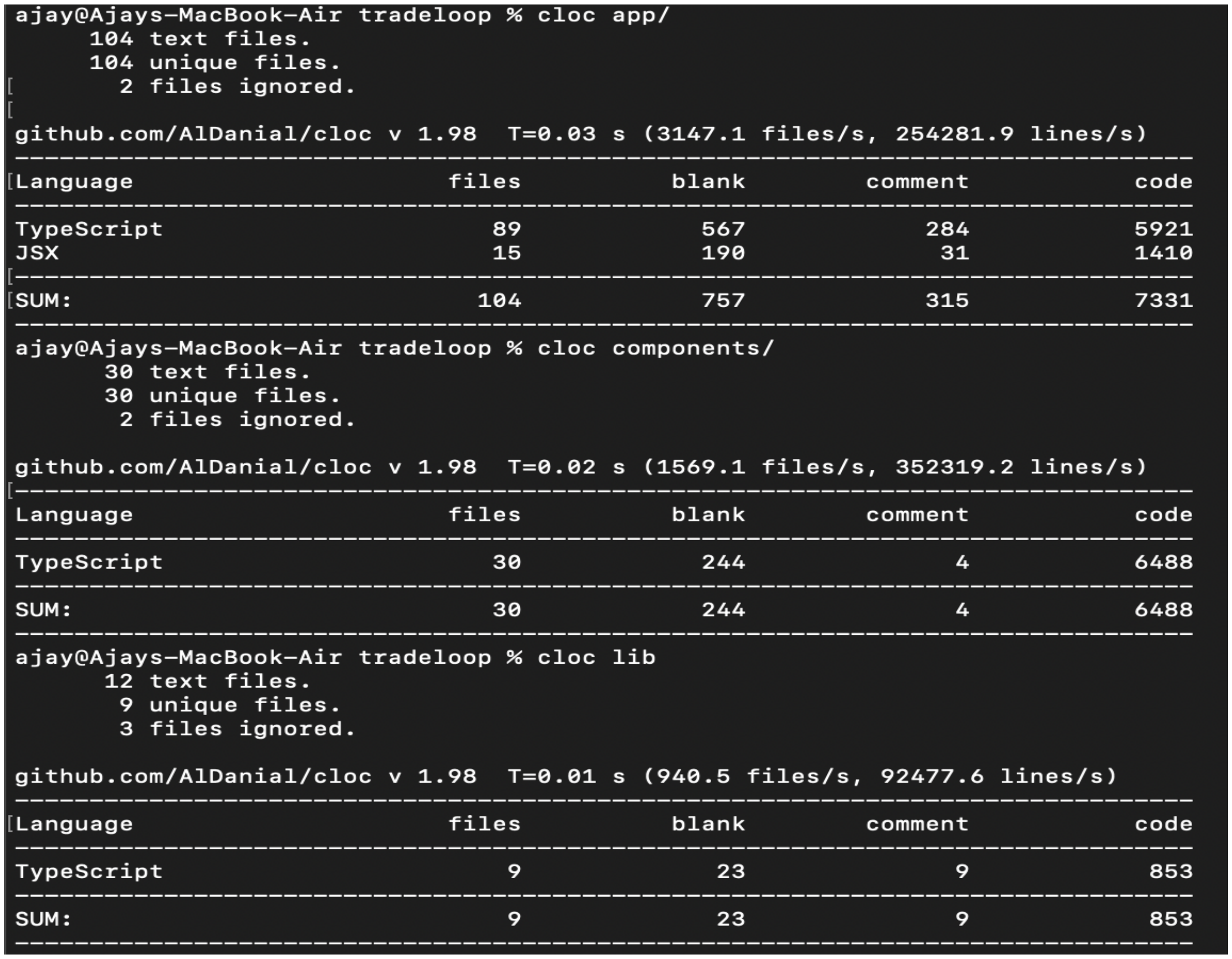
For my showcase project, I have chosen to test "Tradeloop" a trading analytics platform that I have been developing over the past five months. Although Tradeloop is not yet a fully functional web application, its complexity and features make it an ideal candidate for demonstrating my skills in test design and generation. The core functionality of Tradeloop revolves around providing users with a comprehensive dashboard, offering a range of statistics and analytics related to their trading activities. This project has been an excellent opportunity for me to explore and implement the principles of test-driven development.

Hosting:

https://tradeloop-git-testing-cs3150-tradeloop.vercel.app/

Overview:

To determine the lines of code and file count, I utilized an npm package named ‘cloc’. This tool proved user-friendly in installation and operation. In my terminal, I executed ‘npm install -g cloc’, followed by ‘cloc [dir]’ on my three primary web app code directories. The tool helpfully differentiates between blank lines, comment lines, and actual code lines. The results are displayed below:



```
ajay@Ajays-MacBook-Air tradeloop % cloc app/
     104 text files.
     104 unique files.
       2 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.03 s (3147.1 files/s, 254281.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TypeScript                      89            567            284           5921
JSX                             15            190             31           1410
-------------------------------------------------------------------------------
SUM:                           104            757            315           7331
-------------------------------------------------------------------------------
ajay@Ajays-MacBook-Air tradeloop % cloc components/
      30 text files.
      30 unique files.
       2 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.02 s (1569.1 files/s, 352319.2 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TypeScript                      30            244              4           6488
-------------------------------------------------------------------------------
SUM:                            30            244              4           6488
-------------------------------------------------------------------------------
ajay@Ajays-MacBook-Air tradeloop % cloc lib
      12 text files.
       9 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.01 s (940.5 files/s, 92477.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TypeScript                       9             23              9            853
-------------------------------------------------------------------------------
SUM:                             9             23              9            853
-------------------------------------------------------------------------------
```

Architecture of Tradeloop:

Frontend:

- **Framework**: Utilizing Next.js for its efficient React-based structure, which is instrumental in building interactive user interfaces.
- **Styling**: Adoption of Tailwind CSS for a utility-first approach to CSS styling, complemented by Radix UI for crafting accessible components.
- **Linting and Formatting**: Integration of ESLint to enforce code quality standards, and Prettier for consistent code formatting across the project.

Backend:

- **Database**: Implementation of Supabase for robust Postgres database services, including user authentication.
- **Email Services**: Resend Labs for managing email communications, including the deployment of magic links.
- **Backend Services**: Leverage of Vercel for deploying serverless functions efficiently.

Deployment and Hosting:

- **Platform**: Vercel is employed for seamless automatic deployments directly from Github repositories, complemented by CDN for optimized content delivery.
- **Version Control**: GitHub serves as the backbone for source code management, version tracking, and facilitating integration with Vercel's deployment pipelines.

Testing Environment:

In developing my tests, I consciously chose not to utilize the source code for two pivotal reasons. Firstly, I intend to commercialize Tradeloop in the future and wish to keep the source code confidential. Secondly, I sought to challenge myself by learning new methodologies. Given our extensive exposure to Selenium in this course, I opted for an alternative approach. My choice boiled down to unit testing versus end-to-end testing, and I selected the latter, as it does not necessitate direct access to the source code for verification.

After careful consideration, I selected Playwright (version 1.40.0) for its straightforward setup and rapid execution capabilities, which were evident in my preliminary testing. The API of Playwright is notably user-friendly and intuitive. My testing environment consisted of an M1 MacBook Air running macOS 14.0, Node.js version 20.10.0, and TypeScript version 5.3.2. Although my tests were originally written in TypeScript, I will be submitting them in JavaScript to facilitate ease of execution.

Coverage Criteria:

Input Space Partitioning

Testing Documentation:

- At the current state of Tradeloop, we are able to test 7 different pages/components/functionalities. Each one was separated into its own testing file and explained below with multiple coverage criterias for each.

protectedPaths.spec.ts (2 tests)

1. **Protected Paths > Unauthenticated > Protected Paths Redirect to /signin**
    a. Summary: This test verifies that unauthenticated users are redirected to the sign-in page when attempting to access protected paths. It covers the input space partitioning by testing different URL paths (like 'account', 'overview', etc.), representing distinct input categories requiring authentication.
    b. Requirement: Redirect unauthenticated users to signin page
    c. Test cases: visit all 6 protected pages as an unauthenticated user
    d. 100% coverage as it checks all protected routes
2. **Protected Paths > Authenticated > Protected Paths are Accessible**
    a. Summary: This test checks that authenticated users can access protected paths without being redirected. It partitions the input space based on user authentication status and verifies the correct behavior for each of the protected paths, ensuring authenticated access works as intended.
    b. Requirement: Allow authenticated users to access protected pages
    c. Test cases: visit all 6 protected pages as an authenticated user
    d. 100% coverage as it checks all protected routes

signIn.spec.ts (2 tests)

3. **Sign In > Magic Link**
    a. Summary: This test evaluates the passwordless sign-in feature by checking the application's response to both invalid and valid email inputs. It partitions the input space into valid and invalid email categories, ensuring the system correctly identifies and responds to each.
    b. Requirement: Application prompting to check inbox
    c. Test cases: enter invalid email, enter valid email
    d. 100% coverage as all types of inputs are tested
    e. Additional Notes: I've set a strict hourly rate limit for emails (to keep costs down), so running the tests too many times could fail this one.
4. **Sign In > Email & Password**
    a. Summary: This test focuses on traditional sign-in with email and password, covering cases with correct and incorrect credentials. It effectively partitions the input space into two categories: valid and invalid authentication data, ensuring the application appropriately handles each scenario.
    b. Requirement: redirect to overview page after sign in
    c. Test cases: invalid credentials, valid credentials
    d. 100% coverage as all types of inputs are tested

navigation.spec.ts (2 tests)

5. **Navigation > Desktop Sidebar**
    a. Summary: This test assesses the desktop sidebar navigation by clicking through various links (like 'Overview', 'Analytics', etc.) and verifying the correct page loads. It partitions the input space based on different navigation elements, ensuring each link correctly redirects to its respective page on desktop devices.
    b. Requirement: all pages are able to be navigated to
    c. Test cases: each link in navbar is clicked

   d. 100% coverage as all navigation elements are tested from a different page.
   e. Additional Notes: Sometimes the chat page can take too long to load and timeout the test. This is due to it being an abnormally large page due to a tokenizer library. This should happen less frequently in the production url provided above, but was a problem I encountered frequently on local build.

6. **Navigation > Mobile Navbar**
   a. Summary: This test checks the mobile navigation bar by simulating a mobile screen size and navigating through different links. It partitions the input space by screen size (mobile vs. desktop) and navigational elements, ensuring the correct functioning of the mobile navigation interface.
   b. Requirement: all pages are able to be navigated to on mobile
   c. Test cases: each dropdown button for each page is clicked
   d. 100% coverage as we emulate a mobile screen and test all navigation elements from a different page.
   e. Additional Notes: Sometimes the chat page can take too long to load and timeout the test. This is due to it being an abnormally large page due to a tokenizer library. This should happen less frequently in the production url provided above, but was a problem I encountered frequently on local build.

userMenu.spec.ts (4 tests)

7. **User Menu > Navigated to /account**
   a. Summary: This test assesses the functionality of the user avatar component with a dropdown menu, specifically focusing on the settings option and verifying if it correctly navigates the user to the account page. By partitioning the input space to include different options within the user menu, this test ensures that each menu option, particularly the settings, correctly redirects to the respective page, maintaining the integrity of the user interface.
   b. Requirement: navigate to account page
   c. Test cases: click setting in user menu
   d. 100% coverage as we test the specific drawdown item among a few others.
8. **User Menu > Light Mode (& Persist)**
   a. Summary: This test checks the theme switcher functionality in the user menu, ensuring that selecting 'Light Mode' alters the page's theme accordingly and verifying that this preference persists even after the page is reloaded. It effectively partitions the input space into different theme settings, testing the application's ability to not only change the theme but also remember user preferences across sessions.
   b. Requirement: Change to light mode and persist state
   c. Test cases: trigger light mode & refresh page
   d. 100% coverage as we also check if the theme is persisted after page reload.
9. **User Menu > Dark Mode (& Persist)**
   a. Summary: Similar to the light mode test, this one focuses on the 'Dark Mode' option in the theme switcher, confirming that the theme changes to dark mode when selected and that this setting is retained after the page reloads. This test covers another partition of the input space concerning user preferences for

theme settings, ensuring the application's responsiveness and persistence in maintaining these settings.

   b. Requirement: Change to dark mode and persist state
   c. Test cases: trigger dark mode & refresh page
   d. 100% coverage as we also check if the theme is persisted after page reload.

10. **User Menu > Sign Out**
    a. Summary: This test validates the functionality of the sign-out button in the user menu, ensuring that it logs the user out and redirects them to the sign-in page. By testing this, it examines a crucial aspect of user session management, partitioning the input space to include different states of user authentication and session control within the application.
    b. Requirement: redirect to sign in page
    c. Test case: click sign out in user menu
    d. 100% coverage since this is the only way to sign out.

<u>overview.spec.ts (7 tests)</u>

11. **Overview > Database Fallback**
    a. Summary: This test checks the application's ability to fall back to fetching data from the database when the local IndexedDB does not contain the required data, as evidenced by specific console log messages. It partitions the input space based on the data source (IndexedDB vs. database), ensuring that the application correctly chooses the data source depending on the availability of data.
    b. Requirement: database call on fresh browser visit to overview page
    c. Test cases: new browser visit to overview page as signed in user
    d. 100% coverage as it tests both sources of data for application to fetch from.
12. **Overview > IndexedDB Access**
    a. Summary: In contrast to the previous test, this one verifies that on subsequent page loads, the application accesses data from the IndexedDB instead of the database, which is expected to be updated after the initial load. This test is crucial for validating efficient data fetching strategies, ensuring the application minimizes unnecessary network requests by utilizing local data storage effectively.
    b. Requirement: indexedDB access logged in console
    c. Test cases: refresh page after previous test to simulate old browser
    d. 100% coverage as it tests the only local storage being used
13. **Overview > Refresh Button**
    a. Summary: This test ensures that clicking the refresh button on the overview page triggers a forced fetch of data from the database. It checks the application's responsiveness to user-initiated actions for data refresh, partitioning the input space to cover different data fetching scenarios: automated (as in normal page load) versus manual (triggered by the user).
    b. Requirement: database call in network
    c. Test cases: click on the manual refresh button in overview page

d. 100% coverage since the refresh button is only designed to do one thing, force fetch from the database.

14. Overview > Calendar - Date Change

a. Summary: This test examines the functionality of the calendar component, specifically testing whether changing the date triggers data mutation and reprocessing from IndexedDB. It partitions the input space based on different dates selected, ensuring the application dynamically updates and displays relevant data corresponding to the user's selection.
b. Requirement: data mutation logged in console as a result of date change
c. Test cases: change date in calendar component
d. Partial coverage*. Not every single date is tested, so could make an argument that it isn't actually 100%. It would just take an absurdly long time to test, and one selection triggering a data process is enough for our purposes.

15. Overview > Exchange - Account Change

a. Summary: Similar to the calendar test, this one focuses on the exchange account picker, verifying if changing the account triggers data mutation similar to changing dates in the calendar component. This test partitions the input space based on different exchange accounts, ensuring the application's capability to handle data updates and displays based on user-selected accounts.
b. Requirement: data mutation logged in console as a result of exchange account change
c. Test cases: change account in exchange account component
d. Partial coverage*. This test only covers switching to one account. Ideally we would have multiple accounts to switch between and test. The only hindering factor is that linking these accounts have a rate limit of once every 24 hours enforced by my exchange partner, ByBit.

16. Overview > Data Cards

a. Summary: This test confirms the presence and visibility of various data-displaying components (like Portfolio Value, Recent Trades, etc.) on the overview page. It addresses the input space concerning different data visualization elements, ensuring that the application correctly renders all necessary information in a user-friendly manner.
b. Requirement: data card components are visible
c. Test cases: go to overview page and check
d. 100% coverage as all data components are tested

17. Overview > PnL Tooltip

a. Summary: This test verifies the functionality of the PnL (Profit and Loss) tooltip on the chart, ensuring it renders correctly when hovered over. It tests the responsiveness of the UI elements related to data visualization, partitioning the input space to include interactive components like tooltips that provide additional information on demand.
b. Requirements: custom PnL tooltip is rendered when hover on graph component
c. Test cases: go to overview page and hover on graph component
d. 100% coverage as the entire PnL tooltip is tested as one component.

account.spec.ts (5 tests)

18. **Account > Profile Tab > Data Loaded**
    a. Summary: This test checks the account page for correct rendering of user details like the email address, validating the application's ability to retrieve and display user-specific data. It partitions the input space into different sections of user account information, ensuring each section correctly displays the stored data.
    b. Requirement: Email is present and correct
    c. Test cases: go to account page → profile tab and check
    d. Partial coverage. This test only checks for the email and not username. This is because sometimes user's can not have a username. Tests after this input a username and test it though.
19. **Account > Profile Tab > Change Username**
    a. Summary: This test assesses the functionality of changing the username in the profile tab, verifying that the updated username is displayed correctly after a page reload. It tests the application's handling of user input for account modifications, ensuring data persistence and correct display post-update.
    b. Requirement: Username is changed and updated after page reload
    c. Test cases: input new username → submit → reload page → check
    d. 100% coverage as username change update to database is tested, along with reflection on application after page reload. Side note: the username is actually updated optimistically on the client, but that isn't something that we should test due to it not being 'valid' data. It is better to refresh the page, clear optimistic updates, and verify 'real' data.
20. **Account > API Tab > Data Loaded**
    a. Summary: This test evaluates the loading of exchange account data in the API tab, confirming the application's ability to correctly retrieve and display API-related information. It partitions the input space into different sections within the API tab, focusing on the accurate representation of API and exchange-related data.
    b. Requirement: show exchange account name in account → api tab
    c. Test cases: navigate to account → api tab and check
    d. Partial coverage. As mentioned in an earlier test, rate limits prevent us from testing multiple exchanges at this moment, so only one is tested.
21. **Account > API Tab > API Form**
    a. Summary: This test examines the API form for handling invalid and empty inputs, ensuring that appropriate error messages are displayed and that correct network requests are made when valid data is input. It covers a critical partition of the input space related to API management, focusing on input validation and error handling.
    b. Requirement: Reject empty and invalid input
    c. Test cases: submit empty form, submit form with invalid input
    d. Partial coverage. This test covers empty and invalid inputs. A valid input is not covered due to rate limits likely failing the test case every time except the first within a 24 hours period.

22. Account > API Tab > Refresh Exchange Account

a. Summary: This test checks the functionality of the 'Refresh Exchange Account' button, ensuring that the relevant network request is made and the expected result is returned. It tests a specific aspect of API interaction within the application, ensuring that user actions correctly trigger backend processes.
b. Requirement: Network request made to api and confirmation received in UI for that specific exchange account
c. Test cases: Click on exchange account refresh button
d. 100% coverage as we test the only form of input, submit action.

<u>chat.spec.ts (4 tests)</u>

23. Chat > New Chat

a. Summary: This test verifies that a new chat session is initiated upon page load by checking for a specific welcome message, ensuring the chat functionality is properly initialized. It covers the input space concerning the initialization and basic functionality of the chat feature, ensuring the application correctly sets up a new chat session.
b. Requirement: An empty page with the welcome component
c. Test cases: navigate to chat page
d. 100% coverage as the component to be rendered on new chats is tested.

24. Chat > Input Shortcuts

a. Summary: This test assesses the functionality of input shortcuts in the chat, confirming that clicking on predefined messages correctly populates them in the chat input box. It partitions the input space based on different shortcut options, ensuring the chat interface is user-friendly and responsive to quick selections.
b. Requirement: Selecting a shortcut populates the input field with it
c. Test cases: click each shortcut
d. 100% coverage as all 3 input shortcuts are tested.

25. Chat > Send Message

a. Summary: This test checks the functionality of sending a message in the chat, ensuring that it triggers the correct network request to the server. It covers a crucial part of the input space relating to user interactions within the chat feature, ensuring that messages are properly sent and processed by the server.
b. Requirement: network request triggered to specific api route on message send
c. Test cases: send a message
d. 100% coverage as it tests the network request sent to an api route.
e. Additional Notes: Sometimes, the OpenAI API can be slow to respond and the test could timeout, just rerun it.

26. Chat > Share Messages

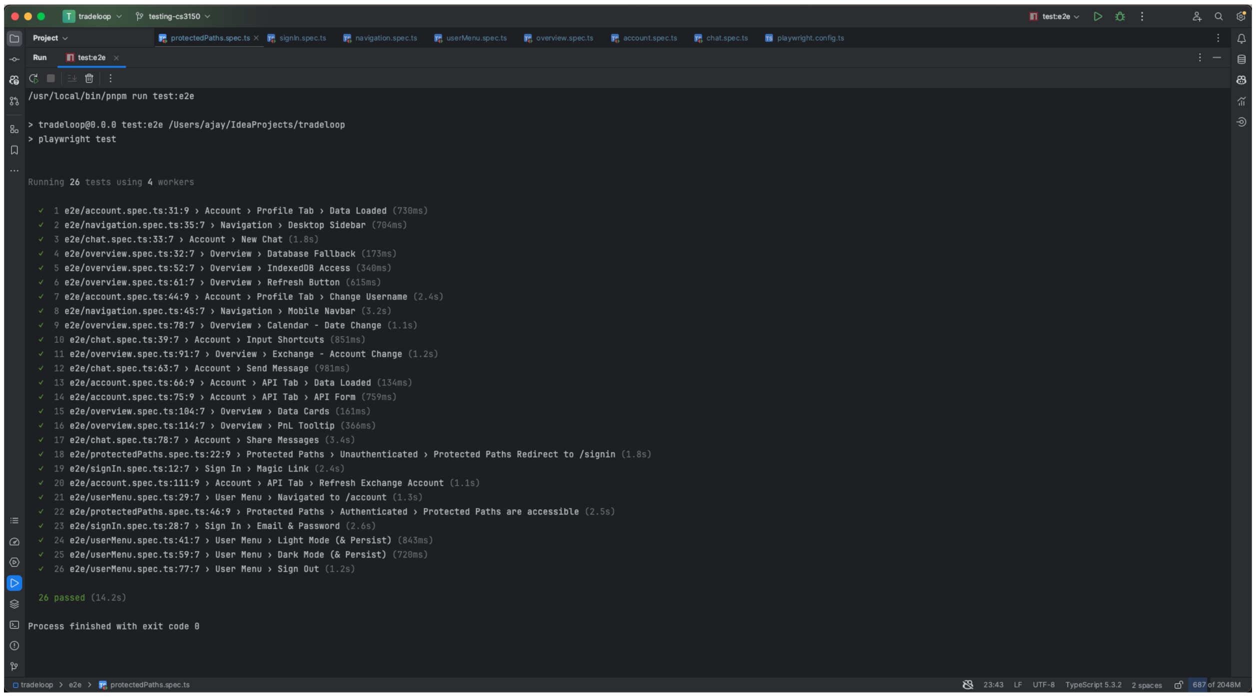
a. Summary: This test evaluates the share functionality in the chat, confirming that shared messages generate a correct link, and when visited, the shared message is visible on the page. It tests the application's ability to handle message sharing, covering the input space related to extending the chat functionality to external sharing and link generation.
b. Requirement: share link copied to clipboard, share page contains sent message

c. Test cases: send message → click share → go to share page and verify
d. 100% coverage as it generates a share link and also verifies contents of it
e. Additional Notes: Sometimes, the OpenAI API can be slow to respond and the test could timeout, just rerun it. Since the shared page is dynamically rendered by serverless, there could be a cold start → page takes too long to load → test times out. Rerunning the test should fix things as the serverless environment for that page is probably warmed up now.

Testing Documentation cont…:

- **Redundancy**: Originally, I had around 50 tests. This was due to each part of the test running as an individual process. For example, there are 6 protected paths that need to be tested. I created 6 tests for the authenticated flow, and 6 more for the unauthenticated flow. But after discovering Playwright's browser context feature, I was able to make those 12 tests into 2. Another way I reduced redundancy was by using the beforeAll feature to sign in the user once instead of writing that logic in each test. Furthermore, I made it so each testing file only used one or two browser contexts. This meant that the browser didn't close and reopen for each test, it just moved onto the next test, making it super fast. An argument could be made that this might interfere with some tests as it could introduce unknown dependency partitions, but I have carefully crafted the tests to prevent this. As a result, each test file is not hard limited to only one browser context, some contain 2. One thing I love about Playwright is that it is a lot easier to get up and running with parallelization compared to Selenium. Additionally, in the userMenu component, I was testing this in both the desktop and mobile view. But since the same exact component is rendered in the desktop and mobile view (nice React feature), I decided to keep only the desktop test. Finally, also in the userMenu component, we could test to validate the user's email displayed in it, but this is retrieved from the same global store that the email we verified in the account page → profile tab, so no need to do it again.
- **Subsumption**: Most of my testing improvements fell under clearing redundancy. There was only one that fitted subsumption and that was the sign out test. Originally, I was checking the auth token being removed and also redirecting to the sign in page. There is actually no need to check the auth token being removed as we previously tested and verified that the user cannot access the sign in page when authenticated. With this in mind, I remove the auth token removal check for the sign out test.
- **Coverage Criteria**: Each coverage method above states whether it's 100% or not. To add to that, some functionality of the application could not be tested as it is still in development. For example, in userMenu there is a keyboard shortcuts button that does nothing at the moment, so it doesn't make sense to test it. Another example is the sign in process. Although google and discord login are currently support, they are a bit buggy at times and even so, would be difficult to test as google and discord have pretty good anti bot mechanisms. Another component that cannot be fully tested would be the chat page as it still requires some prompt engineering to be fully functional. It would also be pretty difficult to test AI output as it is never the same. There is also a settings component in the chat page that was not tested, this component is being removed due to it costing

money for me to allow users to change those settings (using a more expensive API vs using the default cheap one per message).

- **Additional Notes**: All the tests are concise enough that it is easy to trace where the tests halted in case any of them failed. The console should print an arrow indicating which part of the test failed. Also, there is a config file attached to the testing code. If you have a slow machine, I would recommend decreasing the workers (parallel browsers) count to 1 or 2 for best results. Sometimes the event to listen to indexed db or database network calls can fail if you have a slow machine or are running too many browsers in parallel. If you would like to see the tests being run visually, you can disabled headless mode (more performance intensive, would definitely recommend decreasing workers to 2 here). You can decrease workers to 1 if you want to see every single test run individually instead of having multiple browser tabs overlapping. If you are having timeout issues, you can try increasing the timeout to 30 secs (30,000 ms). For best results, I would run the tests in default configuration: headless on, workers 4. Lastly, you can run each test file individually running something like ‘npx run playwright test testfile.spec.js’.
- **Github Repository**: https://github.com/0x-Legend/Showcase-HPD2DZ
- **Github Commit Hash**: 3957873
- **How to run tests locally**: Check GitHub README.md
- **Video of tests being run in headless mode**: Check Github Repository Showcase Testing Video Headless.mov
- **Video of tests being run in head mode**: Check Github Repository Showcase Testing Video Head.mov
- **Screenshot of tests passing**: see below
- 

```
/usr/local/bin/pnpm run test:e2e

> tradeloop@0.0.0 test:e2e /Users/ajay/IdeaProjects/tradeloop
> playwright test


Running 26 tests using 4 workers

  ✓  1 e2e/account.spec.ts:31:9 › Account › Profile Tab › Data Loaded (730ms)
  ✓  2 e2e/navigation.spec.ts:35:7 › Navigation › Desktop Sidebar (704ms)
  ✓  3 e2e/chat.spec.ts:33:7 › Account › New Chat (1.8s)
  ✓  4 e2e/overview.spec.ts:32:7 › Overview › Database Fallback (173ms)
  ✓  5 e2e/overview.spec.ts:52:7 › Overview › IndexedDB Access (340ms)
  ✓  6 e2e/overview.spec.ts:61:7 › Overview › Refresh Button (615ms)
  ✓  7 e2e/account.spec.ts:44:9 › Account › Profile Tab › Change Username (2.4s)
  ✓  8 e2e/navigation.spec.ts:45:7 › Navigation › Mobile Navbar (3.2s)
  ✓  9 e2e/overview.spec.ts:78:7 › Overview › Calendar - Date Change (1.1s)
  ✓  10 e2e/chat.spec.ts:39:7 › Account › Input Shortcuts (851ms)
  ✓  11 e2e/overview.spec.ts:91:7 › Overview › Exchange - Account Change (1.2s)
  ✓  12 e2e/chat.spec.ts:63:7 › Account › Send Message (981ms)
  ✓  13 e2e/account.spec.ts:66:9 › Account › API Tab › Data Loaded (134ms)
  ✓  14 e2e/account.spec.ts:75:9 › Account › API Tab › API Form (759ms)
  ✓  15 e2e/overview.spec.ts:104:7 › Overview › Data Cards (161ms)
  ✓  16 e2e/overview.spec.ts:114:7 › Overview › PnL Tooltip (366ms)
  ✓  17 e2e/chat.spec.ts:78:7 › Account › Share Messages (3.4s)
  ✓  18 e2e/protectedPaths.spec.ts:22:9 › Protected Paths › Unauthenticated › Protected Paths Redirect to /signin (1.8s)
  ✓  19 e2e/signIn.spec.ts:12:7 › Sign In › Magic Link (2.4s)
  ✓  20 e2e/account.spec.ts:111:9 › Account › API Tab › Refresh Exchange Account (1.1s)
  ✓  21 e2e/userMenu.spec.ts:29:7 › User Menu › Navigated to /account (1.3s)
  ✓  22 e2e/protectedPaths.spec.ts:46:9 › Protected Paths › Authenticated › Protected Paths are accessible (2.5s)
  ✓  23 e2e/signIn.spec.ts:28:7 › Sign In › Email & Password (2.6s)
  ✓  24 e2e/userMenu.spec.ts:41:7 › User Menu › Light Mode (& Persist) (843ms)
  ✓  25 e2e/userMenu.spec.ts:59:7 › User Menu › Dark Mode (& Persist) (720ms)
  ✓  26 e2e/userMenu.spec.ts:77:7 › User Menu › Sign Out (1.2s)

  26 passed (14.2s)

Process finished with exit code 0
```

-

Concluding Thoughts: Working on the Tradeloop project has been a great learning experience for me, especially in web application development and test-driven development. Using Playwright for end-to-end testing was crucial for making sure Tradeloop works properly. This testing not only covered all necessary areas but also made the testing process more efficient. It also showed me the importance of having unique identifiers in components, which helped me make the platform better.

The testing covered different parts like login processes, navigating through the app, and specific features like the chat and data visuals. This helped improve these parts of the app. Right now, Tradeloop has seven well-tested pages or features, showing its strong capabilities. But, there are still some parts, like the chat feature and logging in with third-party services, that need more work and testing.

Overall, this project was a good mix of developing new things and doing thorough testing. What I've learned from this will be really useful for me as a software developer and will help me in my future work.

Looking ahead, I want to try out unit testing with Jest and the React Testing Library to focus on the functionality of individual components. I've also heard about TAD (test after development) as an alternative to TDD (test driven development). I'm interested in trying out TAD to see if it makes the development process quicker.