Test Plan and Report – SafeHouse 12/4/2024

The Test Plan and Report includes the following elements

- **System Test scenarios**: Provide a list of system test scenarios. Ideally, the scenarios should relate to specific user stories and associated acceptance criteria.

A. User story 1: As a user, I want search functionality so I can get more personalized listings
B. User story 2: As a renter, I want to able to find affordable listings
    a. Scenario 1: Manual test search bar
        i. used search bar on hundreds of sample listings to ensure that accurate listings can be queried through autocomplete search bar
    b. Scenario 2: Manual test filters
        i. used filters on hundreds of sample listings by clicking each filter option and generating desired results

C. User story 3: As a seller, I want renters to be able to contact me/chat with me through the website
D. User story 4: As a seller, I want to be able to know who I am talking to
    a. Scenario 1: Manual test chat functionality
        i. tested chat feature by setting up two laptops and direct messaging in real time
    b. Scenario 2: Manual test create chat button
        i. created over 50 chats for each renter and landlord in our database to validate correct renter->landlord communication system
E. User story 5: As a landlord, I want to be able to be able to create listings to advertise my property
    a. Scenario 1: Manual test create listings page
        i. tested the created listings page by manually adding sample data (taken from FB groups) into our database and checking for any creation errors
        ii. created over 100 listings of sample data which we verified contained all the accurate information from form data -> database

Unit tests: all stored in the __tests__ directory and use Jest as the testing framework

- Search.test.ts:
    - Search Functionality: Sprint 1, User Story 3: As a renter, I want search functionality so I can find specific listings.
    - User Story (Filtering): As a user, I want the app to remember my housing preferences so that I don't have to enter them every time I open the app, making the housing search faster overall.
    - User Story (Personalized Listings): As a user, I want to be able to filter out listings based on my personal desires.
    - Test is currently not passing.
- Users.test.ts (not passing):
    - Tests the users API route that handles user creation and login
    - It uses Jest to mock dependencies (like database queries and response handling) to simulate different scenarios and verify the API route responds correctly
- Register.test.ts:
    - Tests user registration and Google sign-in functionality
    - Scenarios tested:
        - Initial render: Verifies that the "ChooseUser" page displays correctly.
        - Role selection: Tests that selecting a user role (like "landlord") navigates to the SignUp page.
        - Google sign-in success: Ensures that clicking the Google sign-in button triggers the authentication process, creates a Firestore user, and navigates to the home page.
        - Error handling: Simulates a failure during the Google sign-in process and verifies that errors are handled gracefully without making further calls.
    - Uses mocks to simulate Firebase functions (signInWithGoogle, createFirestoreUser) and Next.js routing (useRouter).
- Geocode.test.ts:
    - Tests the geocode API route used to test listings
    - Scenarios tested:
        - Valid address: When a valid address is provided, checks that getCoordinates() returns the correct latitude and longitude, responds with a 200 status.
        - Missing address: If no address is provided, it checks that the API returns a 400 error with an appropriate message and does not call getCoordinates()
        - Unexpected error: Simulates an error during the coordinate retrieval process, verifying that the API responds with a 500 status and an error message.
- LogIn.test.tsx:
    - Tests user login via Google sign-in

    - Scenarios tested:
        - Initial render: Verifies that the ChooseUser component is rendered initially with a "Go to Sign In" button.
        - Navigating to Sign In: Ensures that clicking the "Go to Sign In" button hides it and displays the Google sign-in button.
        - Google sign-in flow: Simulates a Google sign-in by clicking the Google sign-in button, checks that the signInWithGoogle function is called, and ensures that the user is redirected to the home page (/) upon successful sign-in.
    - Used mocks for useRouter (for navigation) and signInWithGoogle (for authentication), simulating their behavior in the tests.
- Listings.test.tsx:
    - Tests the creation of new listings.
    - Scenarios tested:
        - Successful listing creation:
            - Mocks the coordinate fetching and listing creation functions, simulating a successful creation process.
            - Checks that the correct response status (201) and success message are returned, and verifies that the correct data is passed to both getCoordinates and createListing
        - Coordinate fetching failure:
            - Simulates a failure when fetching coordinates (e.g., if the address is invalid)
            - Checks for the response status is 500 with an appropriate error message
            - Confirms that createListing is not called
        - Unexpected error handling:
            - Mocks an unexpected error during the coordinate fetching step, ensuring the API handles this gracefully with a 500 status and the error message 'Failed to create listing'
            - Confirms that createListing is not called
    - Mocks are used for getCoordinates (for geolocation) and createListing (for saving the listing to the database), simulating their behaviors during the tests
- Chat.test.tsx (not passing):
    - Tests the behavior of the chat function
        - Rendering and selecting a chat:
            - Checks if the chat titles are rendered correctly, and simulates clicking on a chat.
            - After clicking, it checks that the selected chat and its messages are displayed.

        - Calling listenToUserChatsAndMessages:
            - Verifies that listenToUserChatsAndMessages is called with the correct user data when the component is mounted
            - This function is responsible for fetching the user's chats and messages.
        - Displaying placeholder when no chat is selected:
            - Checks that the placeholder message ("No messages to display...") is shown when no chat is selected
        - Scrolling behavior when messages are updated:
            - Tests that the chat scrolls to the bottom when new messages are added or when a chat is selected, by mocking the scrollTop property.
        - Rendering ChatInput component: It ensures that the ChatInput component (used for composing new messages) is rendered when a chat is selected.
    - Mocking is used for the authentication context and chat-related functions.
- Bookmarks.test.tsx: