CSE 115A – Introduction to Software Engineering

Cartrekk

Feb 17 2025

Test Plan and Report

System Test Scenarios

User Story 1: Route Interaction and Social Features

As a user I want to interact with the routes other people as well as my own, and be able to like, comment, and share my routes.

Scenario 1.1: View Personal Routes (Pass/Fail)

1. Login to Cartrekk app
2. Navigate to Profile view
3. User should see a list of their previously recorded routes
4. Each route should display:
    i. Date created
    ii. Distance
    iii. Duration
    iv. Route preview map

Scenario 1.2: Like Route Operation (Pass/Fail)

1. Navigate to any route detail view
2. Tap the like button
3. Like count should increment by 1
4. Refreshing the page should maintain the updated like count

Scenario 1.3: Route Deletion (Pass/Fail)

1. Navigate to Profile view
2. Select any personal route
3. Tap delete button
4. Confirm deletion in dialog
5. Route should be removed from list
6. Database should no longer contain route

User Story 2: Accurate Route Tracking

As a user I need accurate route tracking information

Scenario 2.1: Start New Route (Pass/Fail)

1. Open Cartrekk app
2. Navigate to tracking view
3. Press "Start Tracking" button
4. App should:
    i. Request location permissions if not already granted
    ii. Display current location on map
    iii. Begin displaying distance traveled
    iv. Show elapsed time

Scenario 2.3: Background Tracking (Pass/Fail)

1. Start route tracking
2. Navigate away from app
3. Return to app after 5 minutes
4. Tracking should:
    i. Continue uninterrupted
    ii. Show accurate elapsed time
    iii. Display correct distance
    iv. Maintain complete route polyline

Scenario 2.4: Photo Addition During Route (Pass/Fail)

1. Start route tracking
2. Tap camera button
3. Take photo
4. Photo should:
    i. Upload successfully to S3
    ii. Associate with current route
    iii. Be visible in route details after saving

Unit Tests

Overview

To ensure that the Explore page works as intended, we wrote a series of unit tests focused on the ExploreViewModel. The ViewModel is a key component in our MVVM architecture, acting as the intermediary between the Firestore data layer and the UI. These tests validate that the ViewModel correctly transforms data, updates its state, and handles asynchronous operations such as fetching posts, loading comments, and updating likes.

Scenario 1: Load Friends' Posts (Pass/Fail)

- Purpose:
    - Verify that the ExploreViewModel correctly loads posts from the data layer (simulated by a mock Firestore manager), maps Firestore route data into Post objects, and updates its state accordingly.
- Test Steps:
    1. Configure the mock Firestore manager to return a dummy route with known properties (e.g., docID: "abcd", photos, likes, and a userId of "user1").
    2. Call loadFriendsPosts(userId:) on the ExploreViewModel.
    3. Wait briefly to allow asynchronous updates (e.g. username fetch) to complete.
    4. Assert that the posts array has exactly one post, and that key properties (id, photos, likes, username) match the expected values.
- Pass Criteria:
    - Posts count is 1.
    - The post's id is "abcd".
    - The photos, likes, and initial username are correctly set (in our current implementation, the username remains "user1").

Scenario 2: Add Comment (Pass/Fail)

- Purpose:
    - Ensure that when a comment is added via the ExploreViewModel's addComment method, the corresponding post's comments array is updated accordingly.
- Test Steps:
    1. Set up a dummy Post in the view model with an empty comments array.
    2. Call addComment(postId:userId:username:text:) with a sample comment (e.g., "Great post!").
    3. Assert that the comments array now contains one comment with the expected text.
- Pass Criteria:
    - Comments count increases by 1.
    - The newly added comment's text matches the expected value.

Scenario 3: Load Comments for a Post (Pass/Fail)

- Purpose:
    - Validate that the asynchronous loadCommentsForPost method properly updates the comments for a given post.

- Test Steps:
    1. Place a dummy post into the view model.
    2. Configure the mock to return a dummy comment.
    3. Call loadCommentsForPost(post:) for the dummy post.
    4. Verify that the post's comments array reflects the dummy comment.
- Pass Criteria:
    - The comments count as expected.
    - The comment text (and other properties) match the dummy values provided by the mock.

Scenario 4: Synchronous Like Operation (Pass/Fail)

- Purpose:
    - Confirm that the synchronous likePost(postId:) method immediately increments the like count for a post, ensuring a quick UI response.
- Test Steps:
    1. Create a dummy post with a known initial like count.
    2. Call likePost(postId:).
    3. Assert that the like count is incremented by one.
- Pass Criteria:
    - The like count increases by 1 immediately.

Scenario 5: Update Likes for a Post (Pass/Fail)

- Purpose:
    - Ensure that when the Firestore (via the mock) returns a new like count, the ExploreViewModel's updateLikesForPost method updates the like count appropriately.
- Test Steps:
    1. Set up a dummy post with an initial like count.
    2. Configure the mock Firestore manager to return a specific like count (e.g., 42).
    3. Call updateLikesForPost(postId:).
    4. Assert that the post's like count is updated to the mock value.
- Pass Criteria:
    - The like count matches the value provided by the mock (e.g., 42).

Scenario 6: Asynchronous Like Operation (Pass/Fail)

- Purpose:
    - Verify that the asynchronous version of liking a post (likePost(postId:userId:)) updates the like count based on the backend (simulated by the mock).
- Test Steps:
    1. Create a dummy post with an initial like count.

2. Configure the mock to return an updated like count (e.g., 8) after the like operation.
3. Call the asynchronous likePost(postId:userId:).
4. After waiting briefly, assert that the like count has been updated to the expected value.

- Pass Criteria:
    - The like count equals the expected updated value (e.g., 8).