System Testing Document

for

KeepFit

University of Southern California

Spring 2021

Team 12

Aaron Ly

5110281976

Sajan Gutta

2725022497

Roddur Dasgupta

2659572597

Max Friedman

9342195947

Victor Udobong

9031466326

Devin Mui

2587725548

TABLE OF CONTENTS

TABLE OF CONTENTS 1

1. Preface 3

2. Instructions 4

3.1 Create User Height and Weight are Numbers 6
3.2 Create User Screen Shows When creatingUser true 7
3.3 Login User Screen Shows When loggedIn false, creatingUser false 7
3.4 Calorie Calculation Workout Category Comparison 8
3.5 Calorie Calculation N/A if no category 8
3.6-3.7 Calorie Calculation Weight/Time Comparison 8
3.8 Return to Explore Index Screen from SearchExerciseScreen 9
3.9 Return to Search Pages from Details Screens 9
3.10 Return to Search Index Screen 10
3.11 Video Title is Set 10
3.12 Video Description is Set 10
3.13 Video Workout Category is Set 11
3.14 Video Primary Muscle Group is Set 11
3.15 All Required Video Fields are Set 11
3.16 Liking a Video Reflects in the Database 12
3.17 Unliking a Video Reflects in the Database 12
3.18 Return to Explore Index Screen from SearchLivestreamScreen 12
3.19 Return to Explore Index Screen from SearchWorkoutScreen 13

4. White-box Tests 14

4.1 Rendering App 15
4.2 Create User Missing Field Validation Alert 15
4.3 Create User Existing Username Validation Alert 16
4.4 Login Button Properly Calls loginUser 16
4.5-4.7 Testing loginUser, logoutUser, and createUser Redux Actions 17
4.8 Tracking Screen Renders 17
4.9 Tracking Screen Missing Field Alert 18
4.10 SearchExercisesScreen loads properly 18
4.11 SearchLivestreamsScreen loads properly 18
4.12 SearchWorkoutsScreen loads properly 19
4.13 SearchUsersScreen loads properly 19
4.14-4.15 Testing updateSavedExercises, updateLikedVideos Redux Actions 19

4.16-4.18 Rendering App 20
4.19-4.23 Create Livestream 20

1. Preface

This document is intended to provide a detailed description of the test suite for the first release of the KeepFit application. This application is developed for the customer's benefit, Tian Xie, who has requested the product's testing in this document. This document is oriented explicitly for readership by the customer and developers of the system. It details the reasoning and instructions for running each test case and what the cases do. This robust testing system will ensure reliability for the KeepFit system as development progresses.

Following this preface, readers will be able to find instructions on running the KeepFit test suite. They will also see a description of the overall design and tools used for this test suite. This is followed by detailed reports of all implemented test cases. The README file submitted with this document also contains instructions for running the test cases, though those are within this document.

We encourage readers to pay close attention to this document's test cases and contact the authors with any concerns or questions.

2. Instructions

To run the test suite, use the instructions below. They are also located in the README submitted within this assignment.

Running All Tests:

1. Navigate to the KeepFit directory from your terminal or command line.
2. Make sure you have installed all node_modules.
3. Run npm test.
4. Test case execution will output to the terminal/command line.

Our Testing System Design:

We have decided to use a few different libraries in generating white-box and black-box tests for our testing system. The primary framework for all of our test cases is Jest, which allows us to run several test cases all at once from the command line. Jest also serves as an excellent tool for black-box tests, as we can call our business logic and use expect statements (which are built-in with Jest) to check for the proper outputs given a set of inputs. This is independent of the logic in the application and just checks that it behaves as intended.

We have also used React Native Testing Library for white-box testing, which provides many valuable methods, including render and fireEvent. These allow us to render our UI snapshots for testing, simulate interactions with them, and ensure that state management and our call stack work as intended. Jest also helps us "spy" on specific methods and ensure that our UI's actions call the proper handlers.

We made heavy mocking usage during setting up our tests to replace functions that we weren't testing directly with custom implementations. Lots of mocking occurs in the mock.js file at the base of the KeepFit directory. This was especially important for firebase methods since we didn't want to operate on our actual database. Instead, we utilized firestore-jest-mock, a valuable library, to mock a firebase instance and test a simulated version of a firebase database. This provides mocked versions of many firebase methods to check that our queries are properly building and calling.

Another specific case we had to mock was our Redux store. Anytime we rendered a component, we had to pass in a mocked Redux store to allow that component to find its state correctly. We used redux-mock-store for this. This was very useful for testing state changes in our application and ensuring that render logic worked adequately.

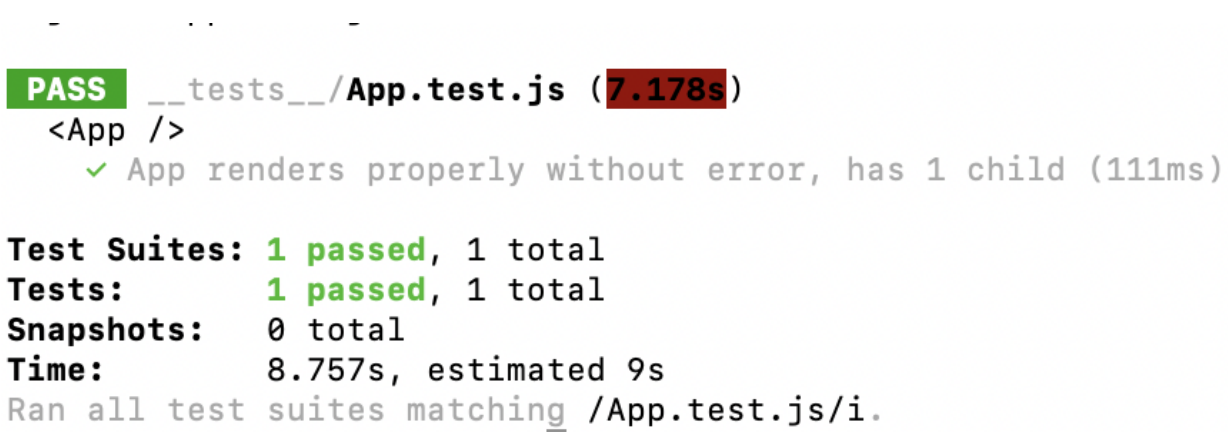
You will find all of our tests in the __tests__ directory, which is at the base of the "KeepFit" directory. Each file has a name describing the type of feature intended to test, followed by the

“.test.js” extension recognized by jest. We have done some configuration in package.json to ensure that our test environment sets up properly. Tests are grouped within files into more specific test suites based on what they check.

3. Black-box Tests

Results of Test Cases:

Unless otherwise stated under the test case, you can assume that it passed if you see a message similar to the screenshot below, stating that the case "passed." As long as there are no red error messages that say "failed," you have successfully run the test.



3.1 Create User Height and Weight are Numbers

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** This case is within describe('Create User Input Validation'). This test case is ensuring that the inputs for a user's weight and height work properly. We want these always to be numbers. We don't care how the inner-workings of the code function, only that we are receiving the proper value from the input. In this case, we try injecting a messed up string "2f5" into each input and receiving the input's new value. We check to see if this is "25" which would be expected behavior, converting input only to a number.
4. **Rationale:** We want to ensure that any future updates we make to this page still ensure that height and weight are numbers. Other parts of the application will potentially have errors when trying to do calculations with those numbers if they aren't. This test is black-box because we are just testing that giving a specific input to the fields leads to a separate output. It doesn't matter what intermediate handling leads to that result.
5. **Discoverable Bugs:** We didn't find any bugs, but if future changes mess up this handling, we will identify the bugs before merging code into production.

3.2 Create User Screen Shows When creatingUser true

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** This case is within describe('Ensuring Proper Auth Rendering Based on Redux State'). This test case renders the root navigator of the application, passing a deliberate initial state as input, in which creatingUser is True in the redux state. It then checks if the CreateUserScreen has rendered by checking for a specific string in the output. If the string is there, we've rendered the screen as intended.
4. **Rationale:** This ensures that the application's authentication flow always prompts new users for extra information beyond their google account. This is, of course, critical, as we may never get that information from the new user otherwise. This test is black-box because we are just testing that giving a specific input state leads to a particular output screen. It doesn't matter what intermediate rendering logic leads to that result.
5. **Discoverable Bugs:** We didn't find any bugs, but if future changes mess up this handling, we will identify the bugs before merging code into production.

3.3 Login User Screen Shows When loggedIn false, creatingUser false

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** This case is within describe('Ensuring Proper Auth Rendering Based on Redux State'). This test case renders the root navigator of the application, passing a deliberate initial state as input, in which creatingUser is False, and loggedIn is False in the redux state. This means we are not creating a user and not logged in, so we should prompt the user to login. It then checks if the LoginScreen has rendered by checking for a specific string in the output. If the string is there, we've rendered the screen as intended.
4. **Rationale:** This ensures that the application's authentication flow always redirects users to log in when necessary. They should never access the authentication-protected parts of KeepFit unless logged in. This is of course, critical, as we must prevent private information, and the other parts of the application aren't designed to render when users aren't logged in. This test is black-box because we are just testing that giving a specific input state leads to a particular output screen. It doesn't matter what intermediate rendering logic leads to that result.
5. **Discoverable Bugs:** We didn't find any bugs, but if future changes mess up this handling, we will be able to identify the bugs before merging code into production.

3.4 Calorie Calculation Workout Category Comparison

1. **Location:** __tests__/track_workouts.test.js
2. **How to Run:** npm test -- track_workouts.test.js
3. **Description:** This test case is within describe('Tracking Screen Tests') and calls the caloriesCalculator function used in the tracking screen. It sets a constant elapsed time and weight to pass into the function. It then calculates the calories for each workout category. We compare the workout categories to each other, expecting them to increase in calories in the order of weightlifting, hybrid, bodyweight, cardio, and HIIT. If this holds, the case will pass.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our calculation function works, regardless of its inner workings. We want HIIT to burn more calories than cardio, which burns more than bodyweight, and so on. This is why we've crafted our inputs and compared the outputs to ensure the workout category is properly impacting calories burned.
5. **Discoverable Bugs:** If we change the calculation function in the future, we will be able to ensure that common knowledge functionality such as this is still intact.

3.5 Calorie Calculation N/A if no category

1. **Location:** __tests__/track_workouts.test.js
2. **How to Run:** npm test -- track_workouts.test.js
3. **Description:** This test case is within describe('Tracking Screen Tests') and calls the caloriesCalculator function used in the tracking screen. It sets a constant elapsed time and weight to pass into the function. It then calculates the calories for an invalid workout category. We expect "N/A" to return, avoiding errors with calculation.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our calculation function works, regardless of its inner workings. We want to display N/A on the UI if the user doesn't select a workout category, ensuring the function returns an expected output. This is why we've crafted our inputs to have a null category and checked to see if the output is "N/A."
5. **Discoverable Bugs:** If we change the calculation function in the future, we will ensure that invalid categories are still properly handled.

3.6-3.7 Calorie Calculation Weight/Time Comparison

1. **Location:** __tests__/track_workouts.test.js

2. **How to Run:** npm test -- track_workouts.test.js
3. **Description:** These test cases are at the end of describe('Tracking Screen Tests') and call the caloriesCalculator function used in the tracking screen. In both we hold the workout category constant. First, we keep time constant but calculate calories burned with low weight and high weight. If the calories burned with a high weight are greater than a low weight, the test will pass. In the second, we hold weight constant and calculate calories burned with a short and a long time. If the calories burned when using a long time are greater than a short time, the test will pass.
4. **Rationale:** These test cases are ensuring that standard expected behavior occurs when calculating calories. We are simply passing useful test inputs and seeing if the outputs behave as expected. When someone is heavier, they burn more calories. When they workout for longer, they also burn more. These things are being tested in these cases.
5. **Discoverable Bugs:** If we change the calculation function in the future, we will ensure that common knowledge functionality such as this is still intact.

3.8 Return to Explore Index Screen from SearchExerciseScreen

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This case is within describe('Explore Screen Tests'). This test case ensures that when we go to the search exercise screen, the back button is both rendered and executes on click.
4. **Rationale:** We want to ensure that we are always able to go back to the previous screen in any future updates to be able to search for other types of content.
5. **Discoverable Bugs:** If the back button isn't rendered or the back function is not passed to the component we could get stuck on the search screen.

3.9 Return to Search Pages from Details Screens

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This case is within describe('Explore Screen Tests'). This test case ensures that when we see the details of an exercise/livestream/workout, the back button is both rendered and executed on click.
4. **Rationale:** We want to ensure that we are always able to go back to the previous screen in any future updates to be able to search for other individual exercises/livestreams/workouts.

5. **Discoverable Bugs:** If the back button isn't rendered or the back function is not passed to the component we could get stuck on the details screen.

3.10 Return to Search Index Screen

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This case is within describe('Explore Screen Tests'). This test case ensures that when we go to see the details of an exercise/livestream/workout, the back button is both rendered and executed on click.
4. **Rationale:** We want to ensure that in any future updates we are always able to go back to the previous screen to be able to search for other individual exercises/livestreams/workouts.
5. **Discoverable Bugs:** If the back button isn't rendered or the back function is not passed to the component we could get stuck on the details screen.

3.11 Video Title is Set

1. **Location:** __tests__/create_videos.test.js
2. **How to Run:** npm test -- create_videos.test.js
3. **Description:** This test case is within describe('Create Video Input Tests') and checks if the user can upload a video without inputting the title. If they cannot move on without inputting it, the test will pass.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our upload video function works, regardless of its inner workings. We want titles to be set for all videos the user uploads.
5. **Discoverable Bugs:** If we change the upload video function in the future, we can still ensure that titles will still be required.

3.12 Video Description is Set

1. **Location:** __tests__/create_videos.test.js
2. **How to Run:** npm test -- create_videos.test.js
3. **Description:** This test case is within describe('Create Video Input Tests') and checks if the user can upload a video without inputting the title. If they cannot move on without inputting it, the test will pass.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our upload video function works, regardless of its inner workings. We want descriptions to be set for all videos the user uploads.

5. **Discoverable Bugs:** If we change the upload video function in the future, we can still ensure that descriptions will still be required.

3.13 Video Workout Category is Set

1. **Location:** __tests__/create_videos.test.js
2. **How to Run:** npm test -- create_videos.test.js
3. **Description:** This test case is within describe('Create Video Input Tests') and checks if the user can upload a video without inputting the workout category. If they cannot move on without inputting it, the test will pass.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our upload video function works, regardless of its inner workings. We want the workout category to be set for all videos the user uploads.
5. **Discoverable Bugs:** If we change the upload video function in the future, we can still ensure that workout category will still be required.

3.14 Video Primary Muscle Group is Set

1. **Location:** __tests__/create_videos.test.js
2. **How to Run:** npm test -- create_videos.test.js
3. **Description:** This test case is within describe('Create Video Input Tests) and checks if the user can upload a video without inputting the muscle group. If they cannot move on without inputting it, the test will pass.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our upload video function works, regardless of its inner workings. We want the primary muscle group to be set for all videos the user uploads.
5. **Discoverable Bugs:** If we change the upload video function in the future, we can still ensure that primary muscle group will still be required.

3.15 All Required Video Fields are Set

1. **Location:** __tests__/create_videos.test.js
2. **How to Run:** npm test -- create_videos.test.js
3. **Description:** This test case is within describe('Create Video Input Tests') and checks if the user can upload a video without inputting all the required fields except the video itself. If they cannot move on without inputting them all, the test will pass.

4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our upload video function works, regardless of its inner workings. We want the all required video fields to be set for all videos the user uploads.
5. **Discoverable Bugs:** If we change the upload video function in the future, we can still ensure that all required video fields will still be required.

3.16 Liking a Video Reflects in the Database

1. **Location:** __tests__/liked_videos.test.js
2. **How to Run:** npm test -- liked_videos.test.js
3. **Description:** This test case is within describe('Liked Videos Tests') and presses the "liked video" button on uploaded videos that were previously unliked. If a set query is called, we know that the liked video was added to our database.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our like function works, regardless of its inner workings. We want the like button to add the specified video to the database for further use if it was previously unliked.
5. **Discoverable Bugs:** If we change the like video function in the future, we will be able to ensure that clicking a video that was previously unliked will add it to the database.

3.17 Unliking a Video Reflects in the Database

1. **Location:** __tests__/liked_videos.test.js
2. **How to Run:** npm test -- liked_videos.test.js
3. **Description:** This test case is within describe('Liked Videos Tests') and presses the "liked video" button on uploaded videos that were previously liked. If a delete query is called, we know that the liked video was removed from our database.
4. **Rationale:** This test case is a black box case, ensuring that a specific functionality of our like function works, regardless of its inner workings. We want the like button to remove the specified video from the database if previously liked.
5. **Discoverable Bugs:** If we change the like video function in the future, we will be able to ensure that clicking a video that was previously liked will remove it from the database.

3.18 Return to Explore Index Screen from SearchLivestreamScreen

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This case is within describe('Explore Screen Tests'). This test case ensures that when we go to the search screen, the back button is both rendered and executes on click.

4. **Rationale:** We want to ensure that in any future updates we are always able to go back to the previous screen to be able to search for other types of content.
5. **Discoverable Bugs:** If the back button isn't rendered or the back function is not passed to the component we could get stuck on the search livestream screen.

3.19 Return to Explore Index Screen from SearchWorkoutScreen

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This case is within describe('Explore Screen Tests'). This test case ensures that when we go to the workout search screen, the back button is both rendered and executes on click.
4. **Rationale:** We want to ensure that in any future updates we are always able to go back to the previous screen to be able to search for other types of content.
5. **Discoverable Bugs:** If the back button isn't rendered or the back function is not passed to the component we could get stuck on the search workout screen.

4. White-box Tests

Coverage Criteria and Strategy:

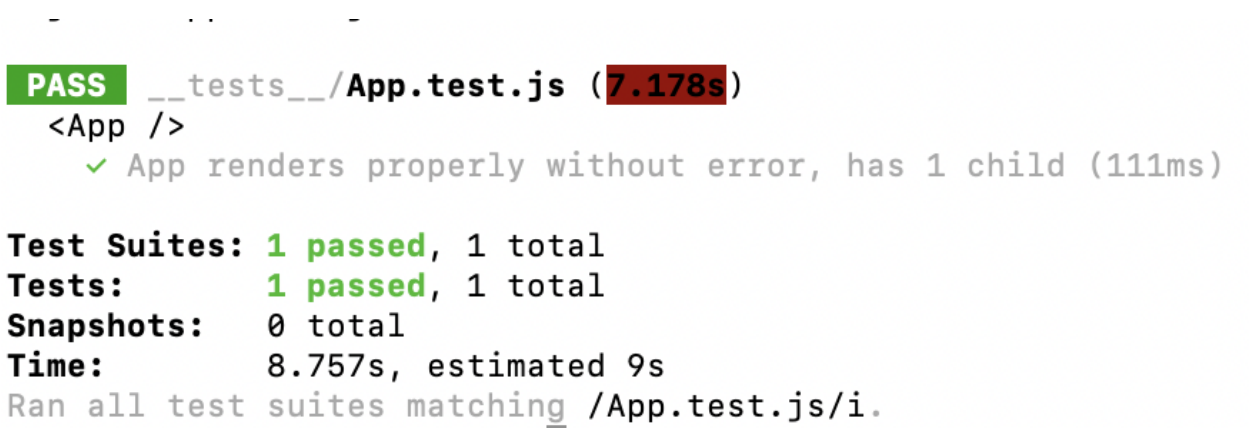
Our goal with our test cases was to cover as much of the essential and testable code as possible in our codebase. We monitored code coverage using Jest, and aimed for 60% or higher on all important files in the code base (most are actually at 100%!). Rather than focusing on the overall code base code coverage, we broke our objectives down based on sections of the code base, as some were more essential than others. There are a few files where we do not have coverages at the goal, but these are intentional and due to a lack of testability or utility in those files. We have explained which files these are and the reason they aren't tested below.

1. **Models Files** - There are a number of files in the models directory of KeepFit that don't necessitate testing. These are merely structured objects that also store firebase collection names. Other than that, they have minimal utility. Thus, there isn't much to test in those files.
2. **Edit Profile Screen** - The coverage here is about 50% since most of this screen is a matter of rendering, for which testing is excessive and won't really catch any errors. The screen doesn't contain any critical business logic.
3. **Main Screen -** The main screen doesn't really need test cases since it is just a navigation container for everything else, containing the navbar at the bottom of the KeepFit UI.
4. **Home Screen -** The home screen doesn't have any content yet, as features it will eventually have are not being implemented until future sprints. Thus, it doesn't need test coverage yet.
5. **Create Index Screen -** This is just a container screen for the create livestreams and upload videos interfaces. Thus, it doesn't need any testing and has low coverage.
6. **RootStackNavigator -** The coverage on this file looks a bit low because it contains lots of Auth handlers which can't be fully tested as they work directly with Google OAuth and Firebase. These have been mocked out and cause coverage to look low. However, most of this file is being tested other than those functions.

Other than the short list of individual files above, we have covered all code with our test cases. We have placed emphasis on addressing the most likely situations to break. This is especially true when working with state management and Redux, as something simple such as accessing an incorrectly named state member can crash the entire application. Our test cases are thorough and will allow us to identify if/when critical KeepFit features break in the future.

Results of Test Cases:

Unless otherwise stated under the test case, you can assume that it passed if you see a message similar to the screenshot below, stating that the case "passed". As long as there are no red error messages that say "failed" you have been successful in running the test.



4.1 Rendering App

1. **Location:** __tests__/App.test.js
2. **How to Run:** npm test -- App.test.js
3. **Description:** This test case is within describe('<App />') and renders the App component that contains our entire KeepFit application. If the App component renders without error, the case will pass.
4. **Rationale:** Tests like this are very common in development as they ensure that new code hasn't prevented the app from properly building. Anytime someone wants to merge in new code, they can run this case to ensure that they haven't broken anything critical and the application can at least still build.
5. **Discoverable Bugs:** This case can identify issues with imports, application structure, etc. We actually uncovered some dependency issues with a few Expo vector icon libraries, which were solved by rebuilding the dependencies for the project to prevent the warnings in the future.

4.2 Create User Missing Field Validation Alert

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** This test case is within describe('Create User Input Validation') and renders the CreateUserScreen. It then tests changing the username field but leaving other fields blank. The case then attempts to create a user with the inputted information and checks to

see if the error handling properly alerts users that they must fill all fields. If the alert method is called, the test case will pass.

4. **Rationale:** We want to ensure that any future updates we make to this page don't break the validation logic. We never want a user to be able to create an account without providing required information. Thus, we've chosen to simulate actual user actions and ensure full robustness against this situation.
5. **Discoverable Bugs:** While we didn't find any now, this case can identify future issues with our validation logic, especially regarding requiring all fields that we request information about.

4.3 Create User Existing Username Validation Alert

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** This test case is within describe('Create User Input Validation') and renders the CreateUserScreen. Unlike case 4.2, it fills all fields. The case then attempts to create a user with the inputted information but with existing information and checks to see if the error handling properly alerts users that the username exists. If the alert method is called, the test case will pass.
4. **Rationale:** We want to ensure that any future updates we make to this page don't break the validation logic for existing usernames. We never want a user to be able to create an account with an existing username. Thus, we've chosen to simulate actual user actions and ensure full robustness against this situation.
5. **Discoverable Bugs:** While we didn't find any now, this case can identify future issues with our validation logic and ensure that usernames remain unique.

4.4 Login Button Properly Calls loginUser

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** This test case is within describe('Login Button Triggers Redux Login and auth reducers work properly'). It renders the LoginScreen and mocks loginUser. An event is then sent to press the login button. We expect our mocked login function to be called as a result. If this is true, the test passes.
4. **Rationale:** We want to ensure that any future updates, especially to the page UI, continue to keep the login handler accessible via the login button. Thus, we are ensuring that the function is called by pressing the login button.

5. **Discoverable Bugs:** While we didn't find any now, this case can identify future issues. During the development process, we encountered a few situations where we replaced our login button and weren't calling the login handler. This will help us realize that quickly.

4.5-4.7 Testing loginUser, logoutUser, and createUser Redux Actions

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js
3. **Description:** These test cases are all within describe('Login Button Triggers Redux Login and auth reducers work properly'). They test the reducers for user authentication in KeepFit. The first one tests loginUser, ensuring that it properly sets the current user, current user id, and loggedIn state to True. The next one tests that logoutUser properly clears the current user id and current user, while setting loggedIn to False. Finally, we check that createUser keeps loggedIn as False but sets creatingUser to True while setting the current user and id. In each case, if the new state returned by the reducer fits the expected criteria, the test will pass.
4. **Rationale:** State management is very difficult in large applications. Particularly, managing authentication can become complex and is essential to protect information. By ensuring that the critical members of our redux state are properly modified by the auth reducers, we will ensure proper state management throughout our application's development.
5. **Discoverable Bugs:** We discovered that createUser was setting loggedIn to True but should keep it at False. This was a great catch and shows why these cases are so important. We need to maintain our state properly and ensure that future reducers fit our authentication flow.

4.8 Tracking Screen Renders

1. **Location:** __tests__/track_workouts.test.js
2. **How to Run:** npm test -- track_workouts.test.js
3. **Description:** This test case is within describe('Tracking Screen Tests') and renders the TrackScreen component that contains the timer and UI for users to track their workout and calories burned. If the component renders without error, the case passes.
4. **Rationale:** Tests like this are very common in development as they ensure that new code hasn't prevented the component from properly building. The TrackScreen relies on some 3rd party code including a timer component. This test will alert us if any external dependencies change and cause errors. It will also alert errors in our own changes.
5. **Discoverable Bugs:** This case is being proactively implemented to raise indication of changes to external dependencies used in the Track Screen.

4.9 Tracking Screen Missing Field Alert

1. **Location:** __tests__/track_workouts.test.js
2. **How to Run:** npm test -- track_workouts.test.js
3. **Description:** This test case is within describe('Tracking Screen Tests') and renders the TrackScreen component that contains the timer and UI for users to track their workout and calories burned. It only fills the workout category field but then presses save, with the input information being incomplete. The function monitors to see if an alert is raised to the user, indicating that our validation has worked to make sure all required fields are full.
4. **Rationale:** This test will ensure that users are only submitting actual complete workouts to the backend. It will save space and ensure that the user doesn't accidentally save workouts without filling out their information.
5. **Discoverable Bugs:** This case is being implemented to identify future bugs in our validation logic that allow a workout to be saved without the required information.

4.10 SearchExercisesScreen loads properly

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This test case is within describe('Explore Screen Tests') and renders the Explore component that contains options to search for different types of workouts, exercises, live streams, and users. It simulates the "Search Exercises" button being pressed and then checks to ensure that the SearchExercisesScreen is loaded in properly.
4. **Rationale:** This test will ensure that when a user wants to search for exercises, the correct screen is loaded and all exercises are loaded unfiltered.
5. **Discoverable Bugs:** This case is being implemented to identify future bugs rendering the wrong screen or not rendering any screen at all when the "Search Exercises" button is clicked.

4.11 SearchLivestreamsScreen loads properly

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This test case is within describe('Explore Screen Tests') and renders the Explore component that contains options to search for different types of workouts,

exercises, live streams, and users. It simulates the “Search Livestreams” button being pressed and then checks to ensure that the SearchExerciseScreen is loaded in properly.
4. **Rationale:** This test will ensure that when a user wants to search for livestreams, the correct screen is loaded and all live streams are unfiltered.
5. **Discoverable Bugs:** This case is being implemented to identify future bugs rendering the wrong screen or not rendering any screen at all when the “Search Live Streams” button is clicked.

4.12 SearchWorkoutsScreen loads properly

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This test case is within describe(‘Explore Screen Tests’) and renders the Explore component that contains options to search for different types of workouts, exercises, live streams, and users. It simulates the “Search Workouts” button being pressed and then checks to ensure that the SearchWorkoutsScreen is loaded in properly.
4. **Rationale:** This test will ensure that when a user wants to search for workouts, the correct screen is loaded and all workouts are loaded unfiltered.
5. **Discoverable Bugs:** This case is being implemented to identify future bugs rendering the wrong screen or not rendering any screen at all when the “Search Workouts” button is clicked.

4.13 SearchUsersScreen loads properly

1. **Location:** __tests__/explore.test.js
2. **How to Run:** npm test -- explore.test.js
3. **Description:** This test case is within describe(‘Explore Screen Tests’) and renders the Explore component that contains options to search for different types of workouts, exercises, live streams, and users. It simulates the “Search Users” button being pressed and then checks to ensure that the SearchUsersScreen is loaded in properly.
4. **Rationale:** This test will ensure that when a user wants to search for other users, the correct screen is loaded and all users are loaded unfiltered, at first.
5. **Discoverable Bugs:** This case is being implemented to identify future bugs rendering the wrong screen or not rendering any screen at all when the “Search Users” button is clicked.

4.14-4.15 Testing updateSavedExercises, updateLikedVideos Redux Actions

1. **Location:** __tests__/authentication.test.js
2. **How to Run:** npm test -- authentication.test.js

3. **Description:** These test cases are all within describe('Login Button Triggers Redux Login and auth reducers work properly'). They test more reducers for user data in KeepFit. The first one tests updateSavedExercises, ensuring that it properly sets the list of saved exercises for a user in redux, so that they may be pulled and rendered in other views. The next one tests that updateLikedVideos properly updates the video ids and video data that a user has liked in the redux state. If these are properly updated in the redux state, they'll be available for rendering in other views and the redux action and reducers will be working properly.
4. **Rationale:** State management is very difficult in large applications. Particularly, managing user information can become complex and is essential to protect information. By ensuring that the critical members of our redux state are properly modified by the auth reducers, we will ensure proper state management throughout our application's development.
5. **Discoverable Bugs:** We discovered better ways to structure our liked video data by closely examining the redux actions. In the future, if new reducers/actions don't work properly for these features, we will be aware and make sure our liked video and saved exercise data is properly stored through redux.

4.16-4.18 Rendering App

1. **Location:** __tests__/profile.test.js
2. **How to Run:** npm test -- profile.test.js
3. **Description:** This test case is within describe('Profile Screen Tests') and has cases to render the Profile Screen, the User Data Screen, and Edit Profile Screens. It ensures that they still render without error and passes if they do.
4. **Rationale:** Tests like this are very common in development as they ensure that new code hasn't prevented screens from properly rendering. These errors cause major issues for users. Anytime someone wants to merge in new code, they can run this case to ensure that they haven't broken anything critical and the relevant screens still render.
5. **Discoverable Bugs:** This case can identify inconsistencies between the Redux state of the application and state information that the profile screens are attempting to render. We don't need much testing of application logic here as these screens mainly serve to display information to end users.

4.19-4.23 Create Livestream

1. **Location:** __tests__/livestream.test.js
2. **How to Run:** npm test -- livestream.test.js

3. **Description:** These test cases are within describe('Input validation and livestream creation works properly') and have cases to validate livestream form input.
4. **Rationale:** Tests like this are very common in development as they ensure that new code does not allow for invalid and potentially dangerous input to be inserted into the database. These errors can cause graphical issues for users. Anytime someone wants to merge in new code, they can run this case to ensure that they haven't broken anything critical and the relevant screens still render.
5. **Discoverable Bugs:** This case can identify future bugs where new inputs or logic allow for invalid inputs. The case can also identify bugs where the form does not submit even though all inputs are valid.