Project Title: KinKin

Authors

- Isita Bagayatkar (014212380)
- Thi Bui (013716313)
- Joshua Lawson (014413464)
- Christina Ng (013960609)
- Eunice Oh (014311076)
- Isabella Piziali (012574289)
- Charlotte Zhuang (013977431)

Preface

- The expected readership of this sprint documentation are the stakeholders, including the students, teaching assistant (TA), and professor.

Instructions

- Run **`npm install`** to install the dependencies
- Then run **`npm test`** to run the tests

Black Box Tests

The blackbox tests are written using mocha js and sinon stubbing.
These tests test the backend. Each test calls a REST endpoint and ensures that it returns the expected status code and result.

Testing setup:
For each model, there exist a set of routes. A testing file corresponding to it exists under ./server/test. Each route is tested for success and failure scenarios to ensure that the endpoint works as expected across different uses.

Individual test files:
Each test file corresponds to all routes associated with a model. At the beginning, a testServer and database are initialized. Before each test, it uses sinon to stub an auth token being passed. After each test, it restores the stub to reset the auth token. After running all tests in the file, it closes the server and mongo connection.

Bugs:

1. Test set up functions throw errors that get counted as failing tests. The afterEach() function resets the verify token function stub after each test; this throws an error each time.
2. The beforeAll() in all files after the first throws an error because the server initialized in each test file is not being closed correctly. This again is counted as a failing test but is actually a problem with the setup.

Running tests:

```
cd server
npm run test
```

This runs all mocha tests in all files in the ./server/test folder

Created tests for the following models:

- User
- Workout task

1. User creation
    a. Location: ./server/test/User.js /PUT
    b. What test case does & how to execute:
        i. The beforeeach function in this file stubs the auth token. This tests user creation by testing the /PUT user object which creates a user object. This test should check whether this endpoint returns 200 success status code and the newly created object

    c. Rationale:
        i. User object creation is a core functionality. Successful creation should return 200. The lack of an auth token should return 403, which this test does not.
    d. Result of executing:
        i. The test returns 200 which matches the expected 200. The test passes.
    e. Was there a bug?
        i. AfterEach() stubbed function restoration throws an error. Not part of blackbox testing. Explained under Bugs.
2. Users: get all users
    a. Location: ./server/test/User.js
    b. What test case does & how to execute:
        i. The beforeeach function in this file stubs the auth token. This tests whether the get /users route gets all user objects in the database. This checks for status code 200 and that an array is returned
    c. Rationale:
        i. Get all users is another functionality used in the all to view all user objects.

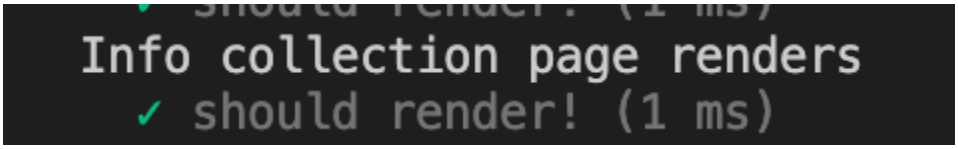
    d. Result of executing:
        i. Passes. The request must have returned 200 and returned an array.
    e. Was there a bug?
        i. None.
3. User: get user by id
    a. Location: ./server/test/User
    b. What test case does & how to execute:
        i. The test case given a valid auth token get's the current authenticated user's object.
    c. Rationale:
        i. This is used to render user profile. Everything client and PT related is associated with User objects. The test should check for return 200 because this id does exist in the database as of now and that the /get returns an object.
    d. Result of executing:
        i. Passes. Returns 200 status code.
    e. Was there a bug?
        i. None.
4. User: get user by id such that the user id does not exist
    a. Location: ./server/test/User
    b. What test case does & how to execute:
        i. The test case is given a valid auth token but a user id that does not exist. Thus, it should return status code 404 because the object does not exist.
    c. Rationale:
        i. To stress test get user by id, giving the endpoint an id that does not exist should result in return status code 404 because the object does not exist.
    d. Result of executing:
        i. Passes. The returned status code is 404.
    e. Was there a bug?
        i. No bugs.
5. User: delete user by id
    a. Location: ./server/test/User
    b. What test case does & how to execute:
        i. The test case is given a valid auth token and a user id for the user object to be deleted. It should return status code 200
    c. Rationale:
        i. User's should be able to delete their account using this endpoint. This test deletes a user that has been created earlier in testing. The test will check for a 200 status code.
    d. Result of executing:
        i. Passes. This returns status code 200.
    e. Was there a bug?
        i. No bugs.
6. User: delete user by id but the user id does not exist

a. Location: ./server/test/User
b. What test case does & how to execute:
    i. The test case is given a valid auth token and a user id that does not exist for the user object to be deleted. It should return status code 404
c. Rationale:
    i. User's should be able to delete their account using this endpoint. This test deletes a user that has been deleted earlier in testing. The test will check for a 404 status code.
d. Result of executing:
    i. Passes. This returns status code 404.
e. Was there a bug?
    i. No bugs.

The same CRUD operation REST endpoints exist across different models: User, Client, PT, workouttask. They all have the same six tests written for them.

White Box Tests

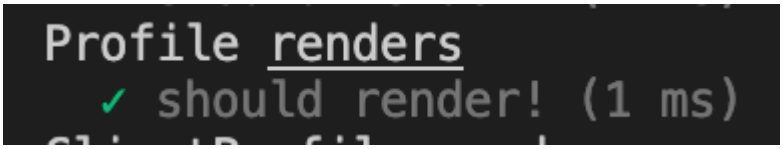
1. Info Collection Page (collection of user info, pt, client etc)
    a. Location: src/App.test.js
    b. What test case does & how to execute:
        i. Test to see if the form for collecting additional user information renders properly, and can gather user information.
        ii. Run using "npm test".
    c. Rationale:
        i. Need to see if the form is functioning properly
    d. Result of executing:



    e. Was there a bug?
        i. Need to be able to verify if user information is able to be collected.
        ii. No
2. Base Profile
    a. Location: src/App.test.js
    b. What test case does & how to execute:
        i. Test sees if user profile renders, which is crucial to functionality–both for showing user their own information, alongside showing user information to other users.
        ii. Run using "npm test".
    c. Rationale:

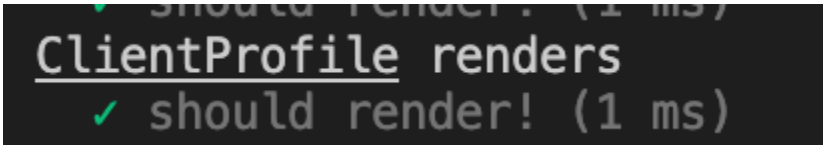
i. Need to be able to test if user can be rendered. Important for “core functionality.”

d. Result of executing:

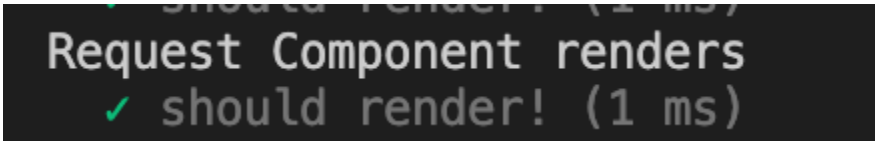


e. Was there a bug?
   i. No bugs!

3. ClientProfile
   a. Location: src/App.test.js
   b. What test case does & how to execute:
      i. Test sees if the specific client profile renders.
      ii. Run using “npm test”.
   c. Rationale:
      i. Need to be able to see if the client profile loads, so others can view profiles.
   d. Result of executing:



   e. Was there a bug?
      i. No
4. Request Component
   a. Location: src/App.test.js
   b. What test case does & how to execute:
      i. Sees if the request component, which is important when sending requests between users, properly renders.
      ii. Run using “npm test”.
   c. Rationale:
      i. Important for functionality of the start of the chat interchange system.
   d. Result of executing:

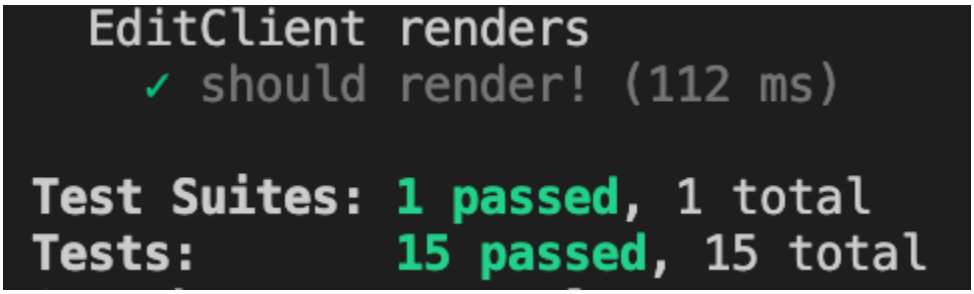


   e. Was there a bug?
      i. No
5. EditClient
   a. Location: src/App.test.js
   b. What test case does & how to execute:
      i. Tests if edit client (which allows the user to edit information about themselves after registering) properly renders.

ii. Run using “npm test”.

c. Rationale:
   i. Need to see if users are properly able to change information about themselves.

d. Result of executing:



e. Was there a bug?
   i. No