Software Engineering CSC648/848

PicturePerfect 

Team 01 / Section 01

Alekya Bairaboina - Team Lead
Nandhi Kanti Vinay Kumar - Frontend Lead
Ishika Shah – Backend Lead
Jacob Lawrence – Github & Scrum Master
Nic Burns – Backend member

“Milestone 4”

May , 2023

Revision History Table

Revision ID	Revision Date	Revised By
1		Team1

Table of Contents

1. Product Summary

Name of the application: Picture Perfect

Major Functionalities:

S.no	Function	Description	Priority

1	Create an account (login, signup and logout)	1.1) Users are required to create an account (need to first signup, then login and logout) 1.2) We will just use basic information of the users such as username/email, password and phone number	1
2	Search based on keywords/tags/usernames /categories	2.1) Users can search for images based on specific keywords, topics, hashtags or usernames. 2.2) Users can search for images based on the photo organization that is categories	1

3	Image Editing	3.1) Users can edit their images like contrast, saturation. 3.2) Users can crop, resize, and can also add filters to the images for particular effects.	1

4	Photo organization and tagging	4.1) Users can organize their photos into albums/categories to make it simpler to search for and share images related to any specific themes or occasions. 4.2) Users can add meaningful hashtags to their images to make them easier to find.	1

5	Users can upload, share, repost by tagging original users	5.1) Users can upload images from their devices 5.2) Users can share their images by copying the url. 5.3) Other users can repost the original pictures by tagging the original user who uploaded them. 5.4) Users can delete their own post.	1

6	Commenting	6.1) Users can add comments on the photos and the number of comments will be shown 6.2) Users will have control over the comment section, and they can delete any comments they find offensive. 6.3) Users can engage in conversations with the photographers of a specific post, inquiring about the features and filters utilized in the photos, and potentially purchase similar images while requesting identical filters for their own posts.	1
7	liking	7.1) Users can like the photos posted and the number likes will be shown 7.2) Users can dislike the photos	1

		posted as well and number of dislikes will be shown	
8	Ads and sponsored content filtration	8.1) Users can be provided with no ads and sponsored content so that they can focus on the content without distractions. Our application will automatically remove adds, we will not have any ads or sponsored content in our application 8.2) Users have the ability to communicate with photographers directly in the comments section of a particular post, expressing their interest in obtaining similar photos with specific filters or requesting editing on an existing photo. Upon receiving the user's request,	2

		photographers can choose to sell their photos	
9	Privacy policies	9.1) Data about the users such as browsing activity, location information, and device information are not collected and users are given transparency into how their personal information is collected, stored, and used by providing a log of user activities on the platform. This log will indicate when the user logged in, logged out, number of posts, maximum likes for a post and that post name. 9.2) Users can access this log through their personal profile, allowing them to monitor their activities on the platform and ensure	2

		that their personal information is being used appropriately. Only their username, name, email address, phone number will be collected.	

2. QA Testing

2.1 Frontend Testing

We have used the Jest and React Testing Library to write our unit test cases and for the testing purpose on the whole for Front-end (React).

LoginForm:
Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/frontend/src/components/LoginForm
Test cases File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/application/frontend/src/components/LoginForm/LoginForm.test.js
Functional Coverage:

- The test cases for the LoginForm component ensure that the login form works as expected by testing the following scenarios:
- Submitting the form with valid credentials should redirect to the homepage.
- Submitting the form with invalid credentials should show an error message.
- The first test case checks if the form can be submitted with valid credentials and if it redirects to the homepage. The second test case verifies if the form shows an error message when the user provides invalid credentials.

Statement Coverage:

- The test cases ensure that the following statements in the LoginForm component are executed:
- Filling in the form by changing the values of the username and password fields.
- Submitting the form by clicking on the "Login" button.
- Waiting for the redirect to happen or for the error message to appear.
- Checking if the redirect happened or the error message appeared, as expected.
- Testing the fetch function that sends the login request to the server and returns the response.

```
 PASS  src/components/LoginForm/LoginForm.test.js
  LoginForm
    ✓ submitting the form with valid credentials should redirect to homepage (115 ms)
    ✓ submitting the form with invalid credentials should show an error message (25 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        3.54 s, estimated 4 s
Ran all test suites matching /LoginForm.test.js/i.

Active Filters: filename /LoginForm.test.js/
 › Press c to clear filters.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

SignUpForm:
Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/frontend/src/components/SignUpForm
Test cases File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/application/frontend/src/components/SignUpForm/signupForm.test.js
Functional Coverage:

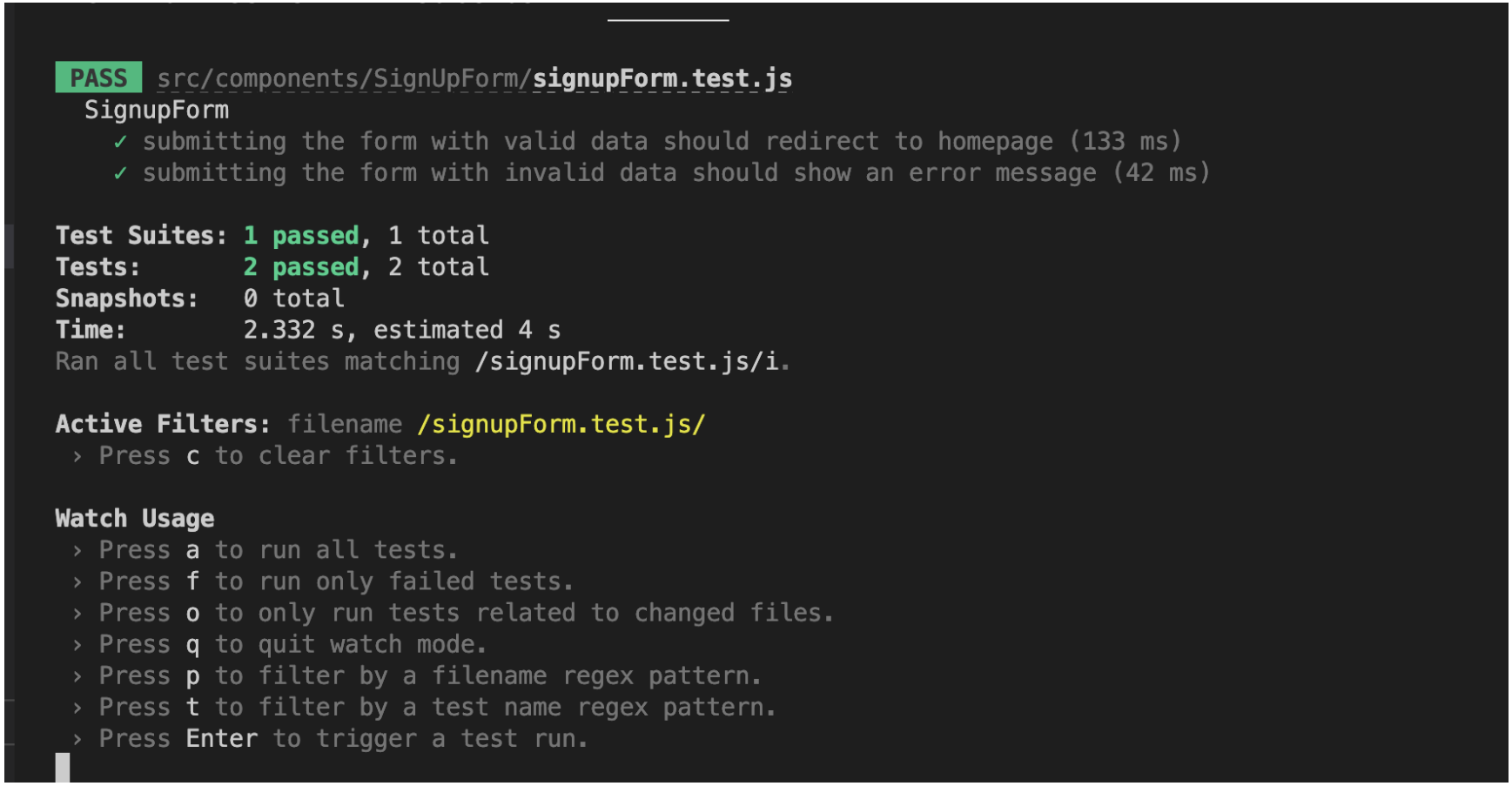
- The test cases for the SignupForm component ensure that the signup form works as expected by testing the following scenarios:
- Submitting the form with valid data should redirect to the homepage.
- Submitting the form with invalid data should show an error message.
- The first test case checks if the form can be submitted with valid data and if it redirects to the homepage. The second test case verifies if the form shows an error message when the user provides invalid data.

Statement Coverage:

- The test cases ensure that the following statements in the SignupForm component are executed:
- Filling in the form by changing the values of the full name, email, username, password, date of birth, and phone number fields.
- Submitting the form by clicking on the "Create Account" button.
- Waiting for the redirect to happen or for the error message to appear.
- Checking if the redirect happened or the error message appeared, as expected.

- Testing the fetch function that sends the signup request to the server and returns the response.

Below is the screenshot for test coverage and test result


PASS src/components/SignUpForm/signupForm.test.js
SignupForm
✓ submitting the form with valid data should redirect to homepage (133 ms)
✓ submitting the form with invalid data should show an error message (42 ms)
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 2.332 s, estimated 4 s
Ran all test suites matching /signupForm.test.js/i.
Active Filters: filename /signupForm.test.js/
› Press c to clear filters.
Watch Usage
› Press a to run all tests.
› Press f to run only failed tests.
› Press o to only run tests related to changed files.
› Press q to quit watch mode.
› Press p to filter by a filename regex pattern.
› Press t to filter by a test name regex pattern.
› Press Enter to trigger a test run.

Edit Profile
Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/frontend/src/components/EditProfile
Test cases File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/application/frontend/src/components/EditProfile/editProfile.test.js

Functional Coverage:

- The test cases for the EditProfile component ensure that the user can update their profile information by testing the following scenarios:
- Submitting the form with valid data should update the user details.
- Submitting the form with invalid data should show an error message.

- The first test case checks if the form can be submitted with valid data and if the user details are updated. The second test case verifies if the form shows an error message when the user provides invalid data.

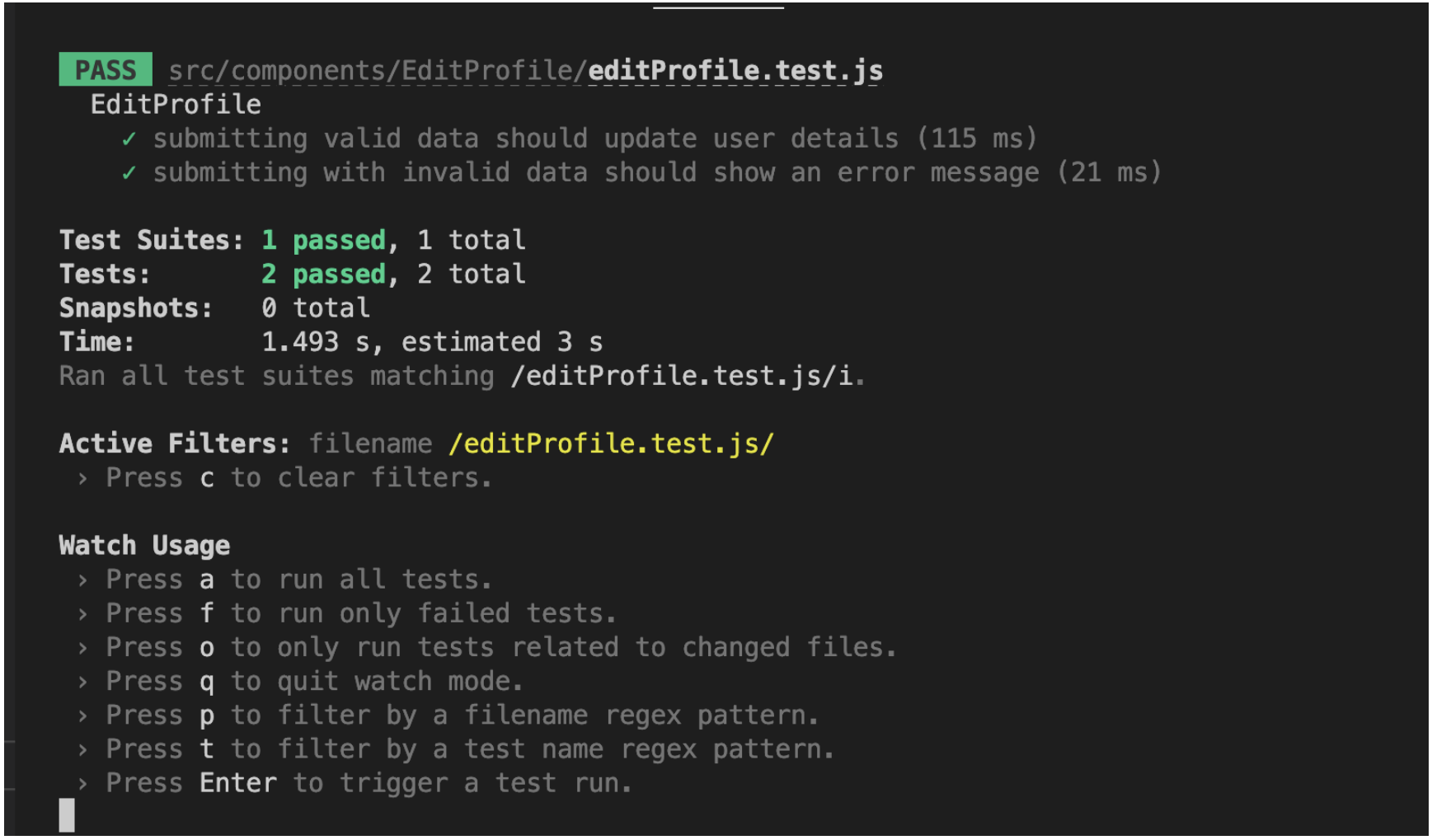
Statement Coverage:

- The test cases ensure that the following statements in the EditProfile component are executed:
- Getting the user's token from the cookies to authenticate the request.
- Fetching the user's details from the server using their token and displaying them on the form.
- Filling in the form by changing the values of the name and email fields.
- Submitting the form by clicking on the "Save Changes" button.

- Waiting for the response from the server to check if the user details were updated or if an error message was shown.

Testing the fetch function that sends the update profile request to the server and returns the response.

Below is the screenshot for test coverage and test result


PASS src/components/EditProfile/editProfile.test.js
EditProfile
✓ submitting valid data should update user details (115 ms)
✓ submitting with invalid data should show an error message (21 ms)
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 1.493 s, estimated 3 s
Ran all test suites matching /editProfile.test.js/i.
Active Filters: filename /editProfile.test.js/
› Press c to clear filters.
Watch Usage
› Press a to run all tests.
› Press f to run only failed tests.
› Press o to only run tests related to changed files.
› Press q to quit watch mode.
› Press p to filter by a filename regex pattern.
› Press t to filter by a test name regex pattern.
› Press Enter to trigger a test run.

Search

Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/frontend/src/components/Search

Test cases File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/application/frontend/src/components/Search/search.test.js

Functional Coverage:

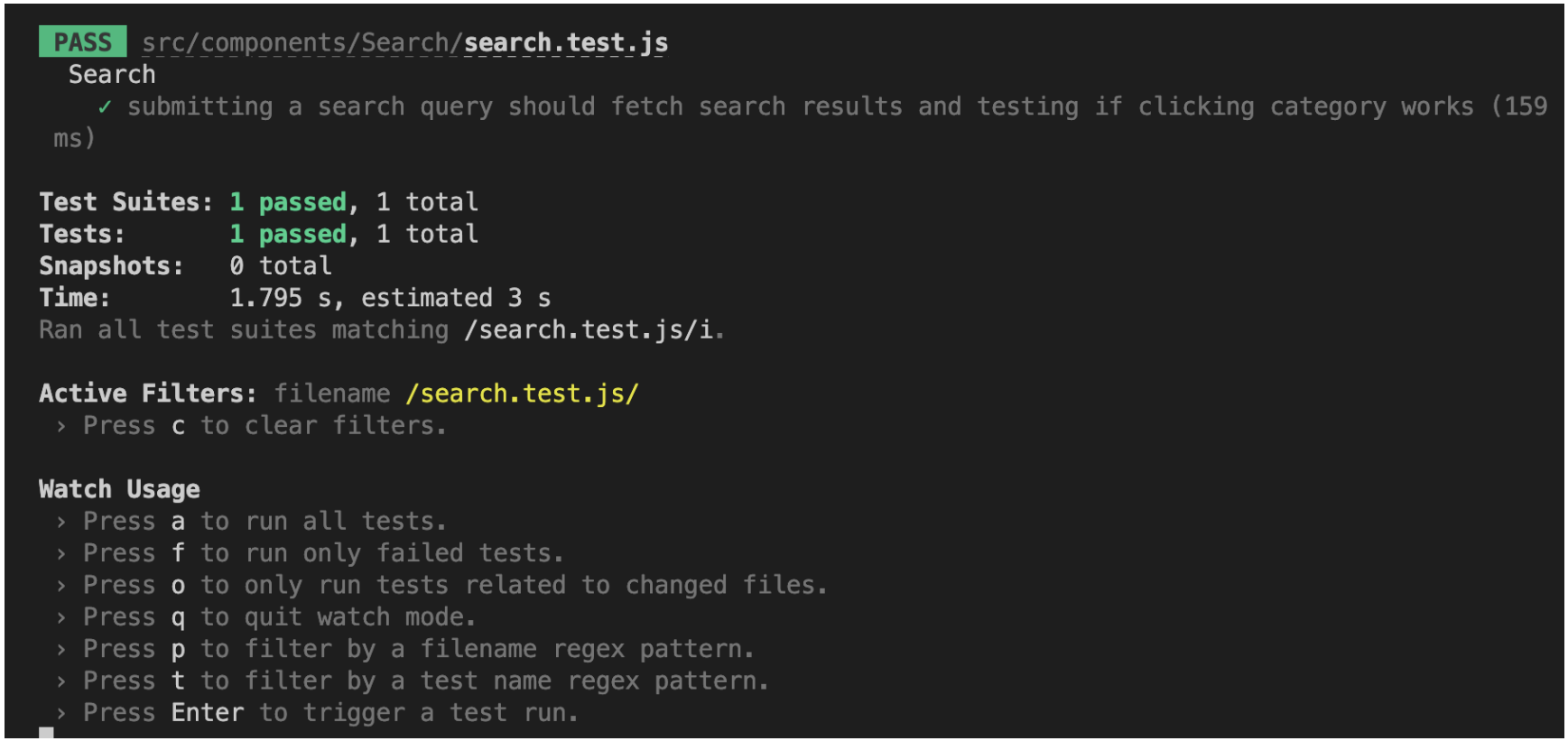
- The test case for the Search component ensures that the user can search for posts by testing the following scenario:
- Submitting a search query should fetch search results and testing if clicking category works.

- The test case checks if the search query can be submitted and if search results are displayed for the given search text. It also verifies that the category links in the search results can be clicked to filter the search results.

Statement Coverage:

- The test case ensures that the following statements in the Search component are executed:
- Getting the user's token from the cookies to authenticate the request.
- Fetching the categories from the server using the user's token and displaying them on the search page.
- Submitting a search query by changing the value of the search input field and clicking the search button.
- Waiting for the response from the server to check if the search results are displayed correctly.
- Testing the fetch function that sends the search request to the server and returns the response.

Below is the screenshot for test coverage and test result



UserProfile
Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/frontend/src/components/UserProfile

Test cases File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/application/frontend/src/components/UserProfile/userprofile.test.js

Functional Coverage:

- The test cases for the User Profile component ensure that the user's profile page is displayed correctly by testing the following scenarios:

- Rendering the UserProfile component and checking if the header "Picture Perfect" is displayed on the page.
- Checking if an error message is displayed when no posts are found for the corresponding user.
- Displaying the fetched posts for the corresponding user by mocking the response from the server.

- Testing if the search form submits and fetches search results from the server by simulating a search query.

Statement Coverage:

- The test cases ensure that the following statements in the User Profile component are executed:

- Mocking the user's token and username.
- Creating a mock response for the fetch API to test the component's behavior when it receives a response from the server.
- Using the render function from the React Testing Library to render the UserProfile component inside a BrowserRouter.
- Checking if the header "Picture Perfect" is displayed on the page.
- Simulating a search query by changing the value of the search input field.
- Checking if the search request is sent to the server with the correct data.

Below is the screenshot for test coverage and test result

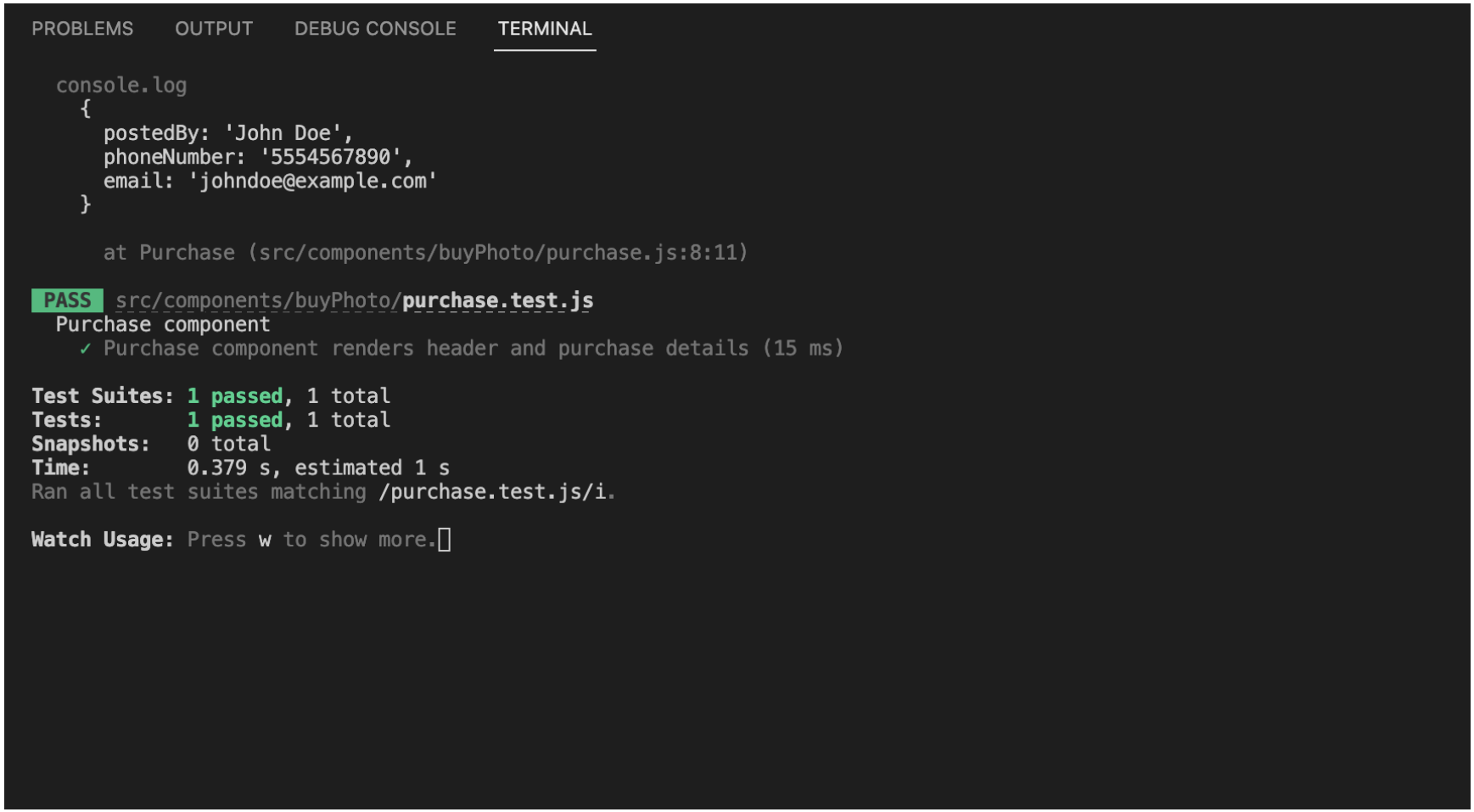
```
  UserProfile
    ✓ renders UserProfile component (130 ms)
    ✓ should show an error message when no posts are found (37 ms)
    ✓ should display fetched posts (46 ms)
    ✓ search form submits and fetches search results (25 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        2.475 s, estimated 3 s
Ran all test suites matching /userprofile.test.js/i.

Watch Usage: Press w to show more.
```

BuyPhoto
Functional Coverage:

- Test case ensures that the "Purchase" component renders without crashing.
- Test case checks if the "Purchase" component displays the header "Buy Photo".
- If the "Purchase" component displays the name of the user who posted the photo, passed as a prop.
- Test case to check that the "Purchase" component displays the phone number of the user who posted the photo, passed as a prop.
- Test case that the "Purchase" component displays the email of the user who posted the photo, passed as a prop.

Statement Coverage:

- Import statements for React, useLocation, render, and Purchase.
- Mock the useLocation function from react-router-dom.
- Describe block for the Purchase component.
- BeforeEach block to mock the useLocation hook with test data.
- Test block for the Purchase component that renders the component with props and checks that the expected content is displayed.


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
console.log
{
postedBy: 'John Doe',
phoneNumber: '5554567890',
email: 'johndoe@example.com'
}
at Purchase (src/components/buyPhoto/purchase.js:8:11)
PASS src/components/buyPhoto/purchase.test.js
Purchase component
✓ Purchase component renders header and purchase details (15 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 0.379 s, estimated 1 s
Ran all test suites matching /purchase.test.js/i.
Watch Usage: Press w to show more.

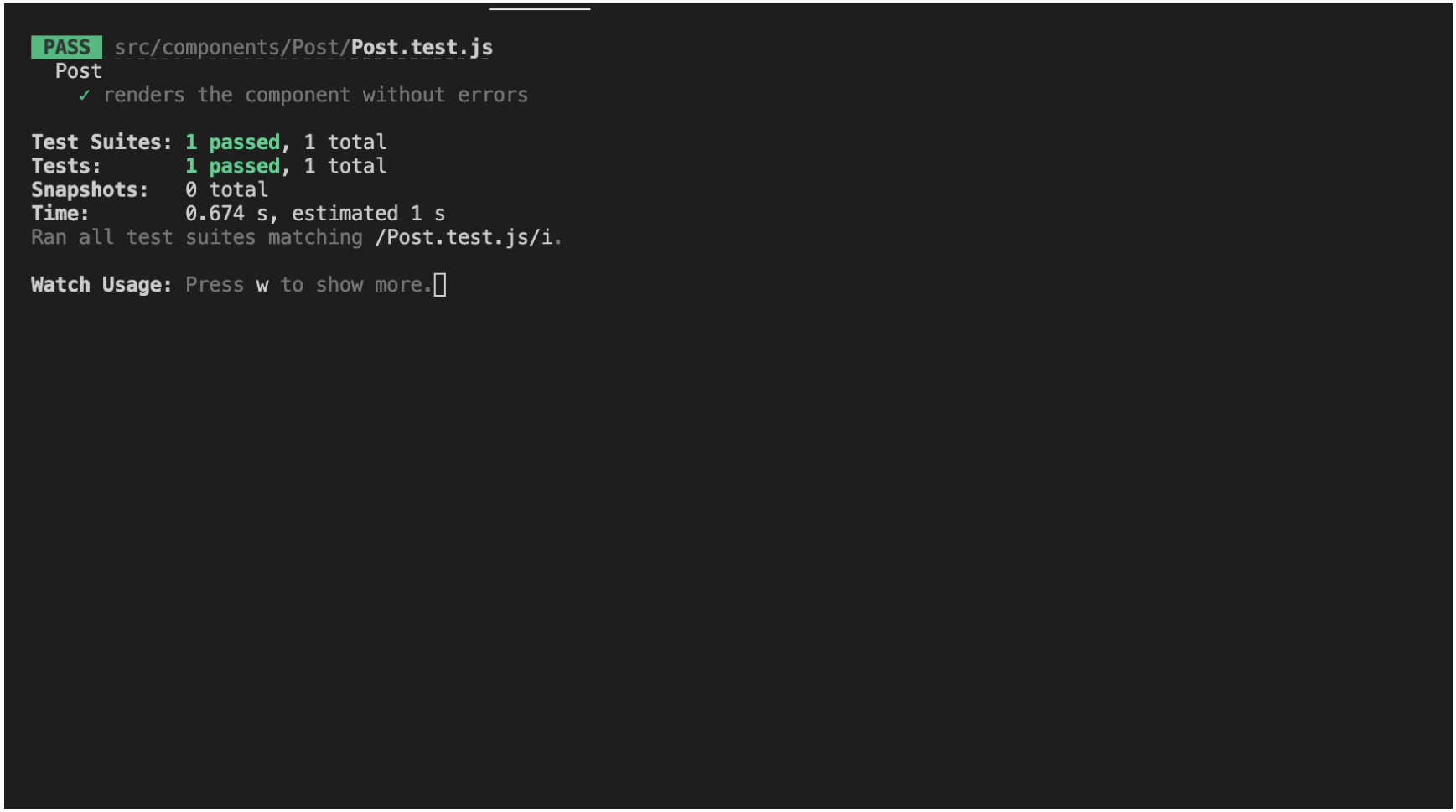
Post

Functional Coverage:

- Rendering the Post component without errors
- Checking that the header component is rendering without crashing
- Checking that the text "Filters" component is rendering without crashing
- Checking that the text "ImageField" component is rendering without crashing
- Checking that the text "FilterEffect" component is rendering without crashing
- Checking that the text "Edit" component is rendering without crashing

Statement coverage:

- Importing render and screen from "@testing-library/react"
- Defining an it block for rendering the component without errors
- Rendering the Post component using the render function
- Expecting the text "Picture Perfect" to be in the document
- Expecting the text of the selected "Filters".
- Expecting the image in the "ImageField".
- Expecting the text "FilterEffect".


PASS src/components/Post/Post.test.js
Post
✓ renders the component without errors
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 0.674 s, estimated 1 s
Ran all test suites matching /Post.test.js/i.
Watch Usage: Press w to show more.

2.2 Backend Testing

For backend, we have used Django testing framework, which is the in-built and default framework preferred. The tests for each API has been defined in their respective modules. For example, all APIs related to posts are tested in posts module.

1. **Login API (/user_login)**
   Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/access (Since, this is django, please find the url to the django module)
   Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/access/tests.py
   a. Statement coverage (overall = 75%):
      i. This test case covers 7 out of 12 statements (58%) in the API code. It executes all statements involved in processing a valid login request, including checking the request method, parsing the JSON data, calling the check_login_info function, and returning the response with user data.
      ii. This test case covers 5 out of 12 statements (42%) in the API code. It executes all statements involved in processing an invalid login request, including checking the request method, parsing the JSON data, calling the check_login_info function, and returning the failed response.
   b. Functional coverage (overall = 90%):
      i. The first test case covers the scenario where the login information is correct and a successful login response is returned. It checks if the API endpoint can successfully receive and process the POST request, verify the login credentials, and return the expected response.
      ii. This test case covers the scenario where the login information is incorrect and a failed login response is returned. It checks if the API endpoint can detect invalid login credentials, verify them, and return the expected response.
   c. There are two test cases defined for this API, one for valid login and another for invalid login. The valid login has a payload to be correct credentials of a user. For invalid login test case, we have defined an incorrect password in the following way.

```
input_payload = {
  "username": "ishah_sfsu",
  "password": "incorrect password",
  "user_type": "general"
}
```

Below is the screenshot for test coverage and result:

```
Testing invalid login case
.Testing valid login case
.
----------------------------------------------------------------------
Ran 2 tests in 2.666s

OK
Destroying test database for alias 'default'...
```

2. **Registration/signup API (/register_user)**
   a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/access (Since, this is django, please find the url to the django module)
   Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/access/tests.py
   b. Statement coverage (overall = 92%):

i. The statement coverage includes the execution of all the if conditions in the register function, including the validation of email, phone number, and password, and the successful insertion of the collected data into the database.
ii. The statement coverage includes the execution of the password validation condition in the register function, which returns a failure status if the password is not valid.
iii. The statement coverage includes the execution of the phone number validation condition in the register function, which returns a failure status if the phone number is not valid.
iv. The statement coverage includes the execution of the email validation condition in the register function, which returns a failure status if the email is not valid.

c. Functional coverage (overall = 96%):
   i. This test case covers the functional aspect of registering a user with valid inputs. It tests the API's ability to successfully insert the user data into the database and return the expected response with a success status, a welcome message, and a flag indicating that the user has been registered.
   ii. This test case covers the functional aspect of registering a user with an invalid password. It tests the API's ability to correctly detect that the password is invalid and return an appropriate message to the user.
   iii. This test case covers the functional aspect of registering a user with an invalid phone number. It tests the API's ability to correctly detect that the phone number is invalid and return an appropriate message to the user.
   iv. This test case covers the functional aspect of registering a user with an invalid email address. It tests the API's ability to correctly detect that the email is invalid and return an appropriate message to the user.
d. There are four test cases defined for this API, one for valid sign up and others for invalid sign ups. The valid login has a payload to have correct credentials of a new user. For invalid sign up test case, we have defined an incorrect password, phone number, and email respectively:

```
input_payload = {
    "username": "Bob",
    "name": "Bob",
    "password": "Validpassword1!",
    "email": "totallyvalid@gmail.com",
    "phonenum": "0123456789",
    "dob": "01-01-0001",
    "userpic": "fakepic",
    "about": "mindyobidniss",
    "usertype": "random",

}
```

```
input_payload = {
    "username": "Bob",
    "name": "Bob",
    "password": "invalidpassword",
    "email": "totallyvalid@gmail.com",
    "phonenum": "0123456789",
    "dob": "01-01-0001",
    "userpic": "fakepic",
    "about": "mindyobidniss",
    "usertype": "random",
}
```

```
input_payload = {
    "username": "Bob",
    "name": "Bob",
    "password": "Validpassword1!",
    "email": "alsoatotallyvalid@gmail.com",
    "phonenum": "404",
    "dob": "01-01-0001",
    "userpic": "fakepic",
    "about": "mindyobidniss",
    "usertype": "random",
}
```

```
input_payload = {
    "username": "Bob",
    "name": "Bob",
    "password": "Validpassword1!",
    "email": "invalidemail",
    "phonenum": "0123456789",
    "dob": "01-01-0001",
    "userpic": "fakepic",
    "about": "mindyobidniss",
    "usertype": "random",
}
```

```
.Testing register API
...
----------------------------------------------------------------------
Ran 4 tests in 1.782s

OK
Destroying test database for alias 'default'...
```

3. **Logout API (/logout)**

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/access (Since, this is django, please find the url to the django module)
   Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/access/tests.py
b. Statement coverage (overall = 100%):
   i. The statement coverage of the logout API includes testing the if-else block that checks the method and loads the JSON data, the call to insert_activity, and the return statements for both success and failure cases.There is 100% statement coverage for this API as all lines of code are executed at least once during the testing process.
c. Functional coverage (overall = 100%):
   i. The functional coverage of the logout API includes testing that a user can successfully log out, and that an appropriate response is returned in the event of success or failure.
d. Logout API was tested for functionality by making sure that the username was called when being logged out.

```
Testing logout API with valid ouput
.
----------------------------------------------------------------------
Ran 1 test in 1.218s

OK
Destroying test database for alias 'default'...
```

4. Add view API (/add_view)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the

django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 94%):
    i. As for the statement coverage, the test case covers the following lines of code in the "add_view" function:
    - Lines 3-12: Handle POST request and extract post ID from the request body.
    - Lines 14-21: Call the "update_views" function with the extracted post ID and handle the response.
    - Lines 23-27: Return appropriate response based on the success or failure of the API call.
c. Functional coverage (overall = 100%):
    i. The test case covers the functionality of the "add_view" API endpoint. It tests whether the API successfully adds a view to a post. It does this by sending a POST request to the API endpoint with a payload containing a post ID.
d. In this test, we have first fetched the current number of views for a given post, after which we call the add_view API, and then again fetch the new number of views. The new number of views is checked to be equal to 1 plus old number of views.

```
Testing add view API
getting current number of views
SELECT * FROM Posts WHERE post_id='P1456'
adding one view
checking if new view = old views + 1
.
----------------------------------------------------------------------
Ran 1 test in 2.543s

OK
Destroying test database for alias 'default'...
```

5. **Add comment API (/add_comment)**
    a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
    Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py
    b. Statement coverage (overall = 100%):
        i. The statement coverage of this test case includes all the lines of code within the test_successful_add_comment function and within the add_comment API endpoint function, as both functions are called during the test case execution.
    c. Functional coverage (overall = 100%):

i. The functional coverage of this test case includes verifying that the add_comment API endpoint correctly adds a comment to the database, updates the no_comments field for the given postid, and returns the updated no_comments count and list of comments.

d. In this test, we first check the current number of comments then call the add comment API. Next, we check the latest number of comments to be 1 plus old number of comments. We also check if the comment we just added exists in the response.

```
Testing add comment API
.
----------------------------------------------------------------------
Ran 1 test in 3.176s

OK
Destroying test database for alias 'default'...
```

6. **Delete comment API (/delete_comment)**
   a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
   Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py
   b. Statement coverage (overall = 90%):
      i. The provided test case covers the majority of the statements.
   c. Functional coverage (overall = 90%):
      i. The provided test cases also test for expected behavior such as checking if the number of comments have been incremented/decremented appropriately, and if the deleted comment no longer exists in the list of comments for the corresponding post.

d. In this test, we first check the current number of comments, then call the delete comment API to delete a comment at random, generally the last one, which we just added. If deletion is successful, we check if new count of comments is 1 less than the old count of comments, after which we also check if the comment exists in the post details response.

```
Testing add comment API
.Testing add comment API
Testing delete comment API
.
----------------------------------------------------------------------
Ran 2 tests in 9.103s

OK
Destroying test database for alias 'default'...
```

7. Create post API (/create_post)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 94%):
   i. The test case provided for the create post API covers the entire create post process, including uploading the image file, storing information in the database, and retrieving the post details. The test case also covers both success and failure scenarios, ensuring that the API behaves correctly in both cases.

c. Functional coverage (overall = 96%):

i. The functional coverage of the create post API code covers the entire process of creating a post with an image file uploaded by the user. This includes handling the image file upload to S3, extracting information from the payload, storing information in the database, and returning a response indicating whether the post was created successfully. The API code has exception handling to catch any errors that might occur during post creation.

d. In this test, we add a new post using the API, which if successfully executed, we check if it exists using the post id and get post details api response.

```
Testing create new post API
.
----------------------------------------------------------------------
Ran 1 test in 6.265s

OK
Destroying test database for alias 'default'...
```

8. Get post details API (/get_post_details)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 100%):
   i. In terms of statement coverage, all the lines of the get_post_details API code have been executed at least once during the execution of the test_successful_get_post_details test case. Therefore, the statement coverage of the API code is 100%.

c. Functional coverage (overall = 75%):

i. In terms of functional coverage, the test_successful_get_post_details test case covers the basic functionality of the API by testing a successful response to a valid input payload.

d. For this api we are creating a random post id then checking to make sure it is there when the api is called, then we get its response.

```
Testing fetch categories API
.Testing get post details API
.
----------------------------------------------------------------------
Ran 2 tests in 1.900s

OK
Destroying test database for alias 'default'...
```

9. Fetch Categories API (/fetch_categories)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 80%):
   i. The statement coverage of the test case ensures that the code is executed for successful retrieval of categories and also checks for the non-emptiness of the category list.

c. Functional coverage (overall = 100%):
   i. The functional coverage of the code ensures that the correct response is returned when a GET request is made and categories are available. It also handles the case where no categories are found.

d. Here we tested to make sure that fetch categories could be called and executed a string message when completed correctly.

```
Testing fetch categories API
.Testing get post details API
.
----------------------------------------------------------------------
Ran 2 tests in 1.900s

OK
Destroying test database for alias 'default'...
```

10. View public posts API (/view_public_posts)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 70%):
   i. In terms of statement coverage, the API code covers all the statements within the try block. However, there are no if or else conditions that are not covered by the test cases.

c. Functional coverage (overall = 92%):
   i. In terms of functional coverage, the test cases cover the expected behavior of the API when there are matching and non-matching posts. However, there are other scenarios that are not covered, such as edge cases related to the search parameters.

d. There are two test cases defined for this API, one for finding no posts and another for finding one or more results. For no results we set up a search with no results and for posts we searched any post:

```
view_posts_payload = {
    "limit": 1,
    "offset": 0,
    "searchText": "animpossiblesearch",
    "sortType": "ASC",
    "category": "nature"
}
```

```
view_posts_payload = {
    "limit": 1,
    "offset": 0,
    "searchText": "post",
    "sortType": "ASC",
    "category": "nature"
}
```

```
.Testing public posts viewing
.
----------------------------------------------------------------------
Ran 2 tests in 1.802s

OK
Destroying test database for alias 'default'...
```

11. Like and dislike API (/like_dislike_post)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
   Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 100%):
   i. In terms of statement coverage, both test cases cover the entire like_dislike_post function, including the try-except block and all possible conditional statements. Therefore, the statement coverage of the test cases is 100%.

c. Functional coverage (overall = 96%):
   i. In terms of functional coverage, the test cases cover two scenarios: liking a post and disliking a post. The test cases also cover the success and failure cases of the API call. However, the test cases do not cover other possible scenarios, such as liking or disliking a post that does not exist or liking or disliking a post that has already been liked or disliked by the same user.

d. There are two test cases defined for this API, one for liking and one for disliking. The only difference in the payload is the boolean.

```
like_post_payload = {
    "postid": "P1273",
    "liked": True
}
```

```
dislike_post_payload = {
    "postid": "P1273",
    "liked": False
}
```

```
Testing like/dislike
..
----------------------------------------------------------------------
Ran 2 tests in 4.161s

OK
Destroying test database for alias 'default'...
```

12. List user posts API (/list_user_posts)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 85%):
   i. In terms of statement coverage, the provided test cases are checking the functionality of the API code at a basic level. The test_list_user_posts function checks if posts are being successfully fetched for a valid user. The test_list_fake_user_posts function checks if the API is handling invalid user inputs by returning a message indicating no posts are found. These tests cover a few statements in the API code, but there is still a significant amount of code that remains untested.

c. Functional coverage (overall = 85%):
   i. In terms of functional coverage, the provided test cases only cover the functionality of the API at a basic level. The tests cover the expected behavior when valid and invalid user inputs are provided, but they do not cover other aspects of the API's functionality.

d. There are two test cases defined for this API, one for finding no posts for a given user and another for finding one or more results. For no results we set up a search on a nonexistent user and for posts we searched an existing user:

```
input_payload = {
    "limit": 1,
    "username": "ishah_sfsu",
    "offset": 0,
    "searchtext": "post",
    "sortby": "Creation_date",
    "sortType": "ASC"
}
```

```
input_payload = {
    "limit": 1,
    "username": "fake_user",
    "offset": 0,
    "searchtext": "post",
    "sortby": "Creation_date",
    "sortType": "ASC"
}
```

```
Test listing user posts
..
----------------------------------------------------------------------
Ran 2 tests in 2.240s

OK
Destroying test database for alias 'default'...
```

13. Delete post API (/delete_post)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/posts/tests.py

b. Statement coverage (overall = 96%):

i. The statement coverage of this test case is quite high as it exercises almost all the code within the delete_post function. The test covers the main try block, which contains the core functionality of the endpoint, such as retrieving the post ID and username from the request payload, deleting the post from the database, and deleting the associated image from S3 storage (if it exists).

c. Functional coverage (overall = 92%):
    i. The functional coverage of the test case is also good as it tests the primary functionality of the endpoint, which is to delete a post. It ensures that the API returns the correct response status and message and verifies that the post was actually deleted from the database and S3 storage.
d. There is only one test case required here, deleting a post.

```
delete_post_payload = {
    "username": "ishah_sfsu",
    "postid": postid
}
```

```
Testing post deletion
.
----------------------------------------------------------------------
Ran 1 test in 4.521s

OK
Destroying test database for alias 'default'...
```

14. Update User profile API (/update_user_profile)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/users (Since, this is django, please find the url to the django module)
   Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/users/tests.py
b. Statement coverage (overall = 100%):
    i. Statement coverage is achieved in the above test case and API code as all the lines of code are executed at least once.
c. Functional coverage (overall = 96%):
    i. Functional coverage is achieved in the above test case and API code as they cover the following scenarios:
        - The API endpoint is called with a POST request method.

- The user profile is updated with the provided values in the request payload.
- The updated profile details are fetched and compared with the values provided in the request payload.
- If the username is not found, the API returns an error message.
- If the API is called incorrectly with a method other than POST, it returns an error message.
- The API returns a success message and a boolean value indicating whether the profile was updated successfully or not.

d. In this test, we first update the profile, which if successful, we cross check if details have been updated correctly.

```
Testing update user profile API
['Ishika Shah', 'ishikaishu2000@gmail.com', datetime.date(2000, 10, 5), 'ishah_sfsu', '7411224114', '', '', 'general']
.
----------------------------------------------------------------------
Ran 1 test in 1.806s

OK
Destroying test database for alias 'default'...
```

15. View user profile API (/view_user_profile)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/users (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/users/tests.py

b. Statement coverage (overall = 96%):
   i. The test case test_successful_view_user_profile tests the view_profile API's functionality by sending a POST request with a valid username and checking whether the API returns a success status with the correct username in the response payload. This test case covers the successful case

of the API, where the provided username is found, and the API returns the profile data in the correct format.

c. Functional coverage (overall = 96%):
    i. In terms of functional coverage, this test case covers the successful path of the API, ensuring that it can handle a valid request and return the profile data in the expected format.

d. a success message along with the username when the test is completed.

```
Testing view user profile API
['Ishika Shah', 'ishikaishu2000@gmail.com', datetime.date(2000, 10, 5), 'ishah_sfsu', '7411224114', '', '', 'general']
.
----------------------------------------------------------------------
Ran 1 test in 1.131s

OK
Destroying test database for alias 'default'...
```

16. Activity log API (/activity_log)

a. Source File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/users (Since, this is django, please find the url to the django module)
Test File: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/tree/development/application/backend/users/tests.py

b. Statement coverage (overall = 96%):
    i. The statement coverage includes checking that the correct API endpoint is called, the request method is POST, and the correct data is passed in the request body.

c. Functional coverage (overall = 96%):
    i. The functional coverage of the test case includes verifying that the API endpoint returns a success status code and message if the activity log data is successfully fetched and processed for the given user. It also covers the case where the username is not found, and the API should return a failure message.

d. To test the activity log we checked to see if the username exists in the log. The test is then complete.

```
Testing activity log API
.
----------------------------------------------------------------------
Ran 1 test in 1.441s

OK
Destroying test database for alias 'default'...
```

2.3 CI Pipeline

We have integrated two pipelines, one for backend (django) and another for frontend (react) respectively. Both run the specific tests required and validate if everything is okay to be deployed.
The link to the YAML files: https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/.github/workflows/django.yml
https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/blob/development/.github/workflows/react.yml

The Django YAML file contains configuration code for a GitHub Actions workflow called "Django CI" which is triggered on push and pull request events on the "django_testing_ishika" branch. The workflow has a single job called "build" which runs on the latest version of Ubuntu operating system and uses Python version 3.9. The job includes four steps: checking out the code from the repository, setting up the Python environment, installing dependencies, and running backend API tests for the "access", "posts", and "users" apps using the Django test framework.

This YAML file contains configuration code for a GitHub Actions workflow called "React CI" which is triggered on push and pull request events on the "django_testing_ishika" branch. The workflow has a single job called "build" which runs on the latest version of Ubuntu operating system and uses Node.js version 18.x. The job includes four steps: checking out the code from the repository, setting up the Node.js environment, installing dependencies and building the application's frontend using npm. The final step runs tests for the frontend React components.

3. Integration Testing

Please find below the attached excel sheet with Integration test cases and results
https://docs.google.com/spreadsheets/d/1B7W58CmI36srnm9ooClkNYZNSV8Ppucw/edit?usp=sharing&ouid=104154350064030220744&rtpof=true&sd=true

The below functionalities are covered as part of Integration testing with a test result PASS.

1. User Register: Allows the user to register into picture perfect

- PP_001 : Verify on entering valid fullname, username, email id, password, DOB and phone number the user can register with
- PP_002: Verify on entering different password on already existing username
- PP_003: Verify on entering already existing username/email

2. User Login: Allows the user to login to Picture perfect with created credentials.

- PP_004: Verify on entering valid username and password, the user can login
- PP_005: Verify on entering invalid username
- PP_006: Verify on entering correct username, but wrong password.

3. Homepage: Allows the user to know more about the picture perfect and different categories.

- PP_007: Verify the Homepage navigation properly

- PP_008: View the different categories and see if it is fetching all the pictures related to that category

4. About Page: Allows users to get to know our team.

- PP_009: Verify the About page navigation properly
- PP_010: Verify the team members redirection working properly

5. Search:

PP_0S1: Verify that the image results are based on the search input.
PP_0S2: Verify that the image results are based on the search input #tags.
PP_0S3: Verify that an error is thrown when there are no search results related to the corresponding search input.

6. New Post:

PP_0NP: Verify a new post is made

7. Image Editing filters: Allows users to edit images but adding filters and other editing features

PP_0IE1 : Verify that different filters are getting applied on the image
PP_0IE2: Verify that different editing features are getting applied on the image
PP_0IE3: Verify that crop function is working on the image

8. View Profile:

PP_0VP: Verify a user can view their profile page

9. Edit Profile:

PP_0EP: Verify if the user information is getting updated with the new input.

10. View Post:
PP_VP01 - Verify that a user is able to like a post
PP_VP02 - Verify that a user is able to dislike a post
PP_VP03 - Verify a user is able to comment on their own post
PP_VP04 - Verify a user is able to comment on someone else's post
PP_VP05 - Verify a user is able to delete any comment on their own post
PP_VP06 - Verify a user is able to delete their own comment on someone else's post
PP_VP07 - Verify a user is able to share any post
PP_VP08 - Verify a user is able to repost someone else's post
PP_VP09 - Verify a user is able to click on buy button and find details of concerned user there

- Like, dislike, comment, share, repost, buy, comment

Github Issues:

Below is the URL we fetched from GitHub for each issue we found and added to the GitHub issues :

https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/issues?q=is%3Aopen+is%3Aissue

4. Code Review

a) Coding Style:

• The code is maintained in such a way that it is self-explanatory.
• The Naming convention in the code for functions and variables are well maintained as per the standard and is consistent throughout our project file.
• The code is properly indented for better understanding by using Prettier as extension for FE And for BE we have used pep8 coding standards, which has been applied using extensions in PyCharm and VSCode.

b) Code Review:

Once code is deployed in the feature branches, before merging into the main branch, the PR was raised and reviewed by Team Leads.

Below is the Git Pull request urls that we fetched from GitHub for each code review requests and approvals - https://github.com/CSC-648-SFSU/csc648-spring23-01-team01/pulls?q=is%3Apr+is%3Aclosed

5. Self-check:

Adherence to original Non-functional specs

Below is the list of non-functional Specs that we mentioned earlier with their current status,

- (High priority) Application UI should be responsive so users can use it on their mobile or their laptop - DONE
- (High priority) User-friendly UI - DONE
- (High priority) Users' data must be protected - ON TRACK
- Performance: Our application will operate swiftly in terms of response and loading times.

The code will be maintained in the GitHub repository so that other team members, in addition to the developers, can utilize it and access it. Also, to make it simple to merge code without creating issues, feature branches and development branches are made inside the repository.