QA & Testing Phase

Front-end:

Contacts.test.js:

Test: "renders without crashing"
This test ensures that the `Contacts` component can render without throwing any errors when provided with the required props. It's a basic test to check the component's rendering capability. In this case, the props are `name`, `src`, and `status`. There's no assertion here because the test will fail if an error occurs during the rendering process.

Test: "renders the Contacts component with name and image"
This test checks if the `Contacts` component correctly renders the name passed to it as a prop. The test first renders the component with the `name` prop set to "John Doe", `src` prop with a placeholder image path, and `status` prop set to "online". Then it asserts that the text "John Doe" is present in the document. This implies that the `Contacts` component is displaying the name as expected. The comment suggests that testing the image could be more complex due to Next.js's `Image` component, so an alternative might be to check for the image's `alt` text or other accessible attributes.

Test: "shows online status when status is online"
This test verifies that the `Contacts` component shows an online status indicator when the `status` prop is "online". After rendering the component with the appropriate props, it looks for an element with the test ID `status-indicator-online`. The presence of this element in the document is then confirmed, which means the component is correctly displaying an online status indicator when the contact is online.

Test: "shows offline status when status is offline"
Similar to the online status test, this one checks if the `Contacts` component shows an offline status indicator when the `status` prop is "offline". The component is rendered with the `status` prop set to "offline" and then the test searches for an element with the test ID `status-indicator-offline`. If this element is found in the document, it indicates that the component is correctly displaying an offline status indicator when the contact is offline.

CreatePost.test.js:

Test Cases

1. "renders without crashing"
This test checks that the `CreatePost` component renders successfully without throwing any errors. There are no explicit assertions here; the test will fail if an error is thrown during the rendering process.

2. "should display the form and input elements"
This test renders the `CreatePost` component and asserts that the text input field is present in the document. It specifically looks for an input with a placeholder text that matches the regular expression `/What's on your mind, John Doe\?/i`, which suggests that the input is personalized for the logged-in user (John Doe in this case). The `i` in the regular expression indicates that the match should be case-insensitive. The test expects this input element to be part of the rendered output.

3. "should display an image upload button"
This test also renders the `CreatePost` component and then asserts that an element with the text 'Photo/Video' is in the document. This text likely represents a button or icon that users can interact with to

upload photos or videos. The test confirms that this element is part of the rendered output, ensuring that the user has the option to upload media with their post.

Feed.test.js:

Test Case:
"renders CreatePost component"
The test case is named to reflect that it should verify the rendering of the CreatePost component within the Feed component. However, the implementation details for this test case are missing. Here's what you would expect to happen in this test:

The test renders the Feed component within the context of both the Redux and NextAuth providers by wrapping it with Provider and SessionProvider respectively. This setup ensures that Feed has access to the mock Redux store and session data it needs to function.

After rendering, the test should query the DOM to find elements specific to the CreatePost component. Since the CreatePost component is likely to be a child component within Feed, the test would confirm its presence to validate that Feed is correctly composing its sub-components.

Header.test.js:

Test Rendering of Header with User Session:
Verifies that the Header component correctly renders the user's name ('John') when a user is logged in, demonstrating proper session handling and user name display.

Test Rendering of Header with Default Icon When User Image Not Available:
Confirms that the Header displays the user's name ('Jane') and a default icon when the user's profile image is not available, indicating proper handling of missing profile images.

Login.test.js:

Render Login Component Without Crashing:
Confirms that the Login component renders successfully with specific button texts.

Render Northeastern University Logo with Correct Properties:
Checks that the Northeastern University logo is displayed with the correct src, height, and width.

Call signIn with 'azure-ad' for Northeastern Account Button:
Verifies that the signIn function is called with 'azure-ad' when the Northeastern Account button is clicked.

Call signIn with 'google' for Google Sign-in Button:
Ensures that the signIn function is called with 'google' when the Google sign-in button is clicked.

Northeastern Account Sign-in Button Styling:
Checks if the Northeastern Account sign-in button has the specified CSS styles.

Post.test.js:

Renders Post Component with Given Props:
Verifies that the Post component renders correctly with the provided mock post data, including the name, timestamp, post text, and image.

Conditionally Renders Image When post.image is Present:
Checks that an image is rendered in the Post component when the post.image property is provided.

Conditionally Does Not Render Image When post.image is Null:
Ensures that no image is rendered in the Post component when the post.image property is null or absent.

Has Like, Comment, and Share Buttons:
Confirms the presence of Like, Comment, and Share buttons in the Post component.

Posts.test.js:

Test Initial State of postSlice:
Verifies that the initial state of the postReducer is an empty array, confirming the default state setup.

Test Adding a Single Post:
Checks if the postReducer correctly handles adding a new post to an initially empty list, ensuring the state updates as expected.

Test Adding Multiple Posts:
Ensures that the postReducer can handle adding multiple new posts to the state, confirming it handles arrays of posts correctly.

Test Updating All Posts:
Verifies that the postReducer can update all posts in the state with a new set of posts, ensuring it replaces the state correctly.

Test Rendering Posts from the API:
Mocks an API call to fetch posts and checks if the Posts component renders these posts correctly, ensuring the component integrates with the Redux store and handles asynchronous data.

RightSidebar.test.js:

Test Rendering RightSidebar Component:
Confirms that the RightSidebar component renders successfully by checking for the presence of the text "Contacts".

Test Display of Icons in RightSidebar:
Verifies that the video, search, and more icons (identified as RiVideoAddFill, BiSearch, and CgMoreAlt) are present in the component.

Test Rendering Contacts for Each Friend:
Ensures that the RightSidebar component renders a contact item for each friend, specifically checking for 'friend_1', 'friend_2', and 'friend_3'.

Test Correct Images Rendered for Contacts:

Checks that each contact in the RightSidebar has the correct image, verifying the src attribute of all images to match a specified URL.

Sidebar.test.js:

Test Rendering of Sidebar with User Session:
Verifies that the Sidebar component renders the user's name ('John Doe') when a user is logged in, demonstrating proper session handling.

Test Rendering with User Name and Default Icon When Image Not Available:
Confirms that the Sidebar displays the user's name ('Jane Doe') and a default icon when the user's profile image is not available.

Test Display of Sidebar Items:
Ensures that the Sidebar component correctly displays various sidebar items like 'Friends' and 'Groups', indicating proper rendering of its elements.

Test Sidebar Component Against Snapshot for Consistency:
Checks if the current render of the Sidebar component matches a saved snapshot, verifying that the component's structure remains consistent over time.

Sidebaritems.test.js:

Test Rendering of Sidebaritem with Icon and Value:
Confirms that the Sidebaritem component renders correctly with the given props, displaying both the text 'Groups' and the associated icon.

Test Rendering of the Provided Icon:
Verifies that the Sidebaritem component correctly renders the provided icon (MdGroups), ensuring the icon component is integrated properly.

Test Rendering of the Text Value:
Checks if the Sidebaritem component correctly displays the text value ('Groups'), indicating proper handling of text properties.

Test Sidebaritem Component Against Snapshot for Consistency:
Ensures that the current render of the Sidebaritem component matches a saved snapshot, verifying that the component's structure and appearance remain consistent over time.

Back-end:

PostControllerTest:

The PostControllerTest class is a set of JUnit tests for the PostController class in a Spring-based application designed for a social network. The test class uses the Mockito framework for mocking the PostService dependency and focuses on testing two main functionalities: adding a new post and retrieving posts.
In the testAddPost method, a POST request is simulated to the "/api/v1/post" endpoint with specified parameters such as post content, email, name, file, and profile picture. The test verifies that the controller

correctly handles the request by mocking the behavior of the postService.addPost method and asserting the expected response status (OK), content type (JSON), and the content itself.
In the testGetPost method, a GET request to the "/api/v1/post" endpoint is simulated. The test mocks the behavior of the postService.getPost method to return a list of mock posts. The assertions then validate that the controller correctly handles the request by checking the response status, content type, and the structure and values of the returned JSON array containing posts.

PostEntityTest:

The PostEntityTest class contains a set of JUnit tests for the PostEntity class, which likely represents an entity in a social networking application. Here's a brief explanation of each test:

- No-args Constructor Test (testNoArgsConstructor):
  - It checks that the no-args constructor of PostEntity creates an instance that is not null.
- All-args Constructor Test (testAllArgsConstructor):
  - It verifies that the all-args constructor correctly sets the values of the PostEntity fields when creating an instance.
- Lombok Builder Test (testBuilder):
  - This test ensures that the Lombok-generated builder method properly constructs a PostEntity instance with the specified values for its fields.
- Equals and HashCode Test (testEqualsAndHashCode):
  - It checks that the equals method correctly identifies two instances of PostEntity as equal if their fields are equal.
  - It also verifies that the hashCode method produces the same hash code for equal instances.
- ToString Test (testToString):
  - This test validates that the toString method of PostEntity generates a string representation of the object in the expected format. It compares the generated string with a manually constructed expected string.

PostServiceImplTest:

This is a JUnit test class for the PostServiceImpl class, which is part of a social network application. The tests use the Mockito framework for mocking dependencies.

- Initialization:
  - The test class initializes Mockito annotations and sets up the necessary mocks for testing.
- testAddPost():
  - Creates a test Post object and a corresponding PostEntity.
  - Mocks the behavior of the postEntityRepository's save method to return the created PostEntity with an assigned ID.
  - Calls the addPost method of the postService with the test Post object.
  - Verifies that the service method returns the expected result and that the repository's save method was called once with any PostEntity.
- testGetPost():
  - Creates two test PostEntity objects.
  - Mocks the behavior of the postEntityRepository's findAll method to return a list containing the two test entities.
  - Calls the getPost method of the postService.
  - Verifies that the service method returns a list of Post objects with the expected properties.
  - Verifies that the repository's findAll method was called once.

PostTest:
This is a JUnit test class for the Post class, which represents a post in a social network application. The tests cover the no-argument constructor, all-argument constructor, builder pattern, equals and hashCode methods, and the toString method of the Post class.

- testNoArgsConstructor():
    - Tests the no-argument constructor by creating a Post object and asserting that it is not null.
- testAllArgsConstructor():
    - Tests the all-argument constructor by creating a Post object with specific attribute values.
- Asserts that the object is not null and that its attributes are correctly set.
- testBuilder():
    - Tests the builder pattern by creating a Post object using the builder and specifying attribute values.
    - Asserts that the object is not null and that its attributes are correctly set.
- testEqualsAndHashCode():
    - Tests the equals and hashCode methods by creating two Post objects with the same attribute values.
    - Asserts that the two objects are equal and have the same hash codes.
- testToString():
    - Tests the toString method by creating a Post object with specific attribute values.
    - Defines the expected string representation of the object and asserts that toString produces the expected result.