

REPORT 3 Method Analysis and Testing

Notebridge5

<i>Method explanation and description.....</i>	<i>2</i>
<i>a)UserPostLikeController.....</i>	<i>2</i>
<i>b)UserPostCommentControl.....</i>	<i>2</i>
<i>c)UserController.....</i>	<i>3</i>
<i>d)PostController.....</i>	<i>3</i>
<i>e)FollowController.....</i>	<i>4</i>
<i>Implications after testing.....</i>	<i>5</i>
 <i>Black box testing.....</i>	 <i>6</i>
 <i>User story testing.....</i>	 <i>7</i>
 <i>Unit testing.....</i>	 <i>8</i>
 <i>Integration and Manual Testing.....</i>	 <i>12</i>

1- Method explanation and description

In our project, we used a variety of methods and endpoints to enable various functionality such as like posts, leaving comments, following users, user login and sign-up, and uploading material. These elements are necessary for establishing an engaging and interactive platform, allowing users to smoothly conduct essential functions like interacting with others and sharing material, hence improving their overall experience with the application.

1) UserPostLikeController

-> The UserPostLikeController includes a constructor that sets up the UserPostLikeService, which contains the logic for managing likes. It handles problems such as ClassNotFoundException and SQLException to guarantee appropriate configuration and database connectivity.

-> The @POST function handles HTTP POST requests to /likes by receiving a LikeDTO object with postId and userId. It uses likePost from UserPostLikeService to register the like and returns a 201 Created status if successful. In the event of a SQLException, it produces a 500 Internal Server Error.

-> The @DELETE function handles HTTP DELETE requests for /likes/{postId}/{userId}. It extracts the postId and userId using @PathParam and then runs unlikePost from UserPostLikeService to remove the like. A successful operation produces a 204 No Content status, but a SQLException generates a 500 Internal Server Error.

2) UserPostCommentControl

-> The `UserPostCommentController` constructor sets up the `UserPostCommentService`, which includes the logic for managing comments. It handles problems like `ClassNotFoundException` and `SQLException` to guarantee appropriate configuration and database connectivity.

-> The `@POST` function handles HTTP POST requests to the `/comments` endpoint and accepts a `CreateCommentDTO` object with comment information. It uses `UserPostCommentService`'s `createComment` method to add the comment and provides a 201 Created status when successful. If a `SQLException` happens, it generates a 500 Internal Server Error.

-> The `@GET` method, translated to `/comments/post/{postId}`, accepts HTTP GET requests to get comments by `postId`. To obtain comments, it calls `getCommentsByPostId` from `UserPostCommentService` after extracting the `postId` using `@PathParam`. If successful, it returns the comments with a status of 200 OK; otherwise, a 500 Internal Server Error is issued in the event of a `SQLException`.

-> The `@DELETE` function handles HTTP DELETE requests for `/comments/post/{commentId}`. It uses `@PathParam` to obtain the `commentId` and then calls `deleteComment` from the `UserPostCommentService` to remove it. A successful operation produces a 204 No Content status, but a `SQLException` generates a 500 Internal Server Error.

3) UserController

-> The `UserController` class has a constructor that creates an instance of the `UserService`, which performs the business logic for user management. This configuration guarantees that the controller is ready to efficiently manage user-related actions and handle errors, hence maintaining system stability.

-> The `@POST` method, which uses the `/signup` URL, handles user registration requests. It receives a `CreateUserDTO` object containing user information and registers the user using the `UserService`'s `createUser` function. When registration is successful, it returns a 201 Created status.

-> The `@GET` method returns all users. It uses `getAllUsers` from `UserService` to retrieve a list of `User` objects, transforms them to `GetUserDTO` objects, and returns the list. This approach provides clients with a thorough list of all users in the system.

-> The `@GET` function, with the `/id` path, obtains a specific user by their ID. To obtain user information, it calls `getUserById` in `UserService` after extracting the ID using

@PathParam. If the user is located, it returns the user information with a 200 OK status; otherwise, it returns 404 Not located.

-> User authentication is handled by the @POST method, which uses the /login path. It receives a LoginUserDTO with the email and password, then uses GetUserToken in UserService to produce a token, which it returns if authentication is successful. If the authentication fails, it returns a 401 Unauthorized status.

4) PostController

-> The PostController class has a constructor that creates an instance of the PostService, which performs the business logic for posting. This configuration guarantees that the controller is ready to efficiently manage post-related actions and handle errors, hence maintaining system stability.

-> The @POST function accepts HTTP POST requests for the /posts endpoint. It receives a CreatePostDTO object containing post information and invokes the PostService's createPost function to create a new post. When successful, it returns a 201 Created status.

-> The @GET function returns all posts. It uses PostService's getAllPosts method to obtain a list of Post objects, transforms them to GetPostDTO objects, and returns this list. This technique provides clients with a full list of all postings in the system.

-> The @GET function using the /{id} path gets a specific article based on its ID. It extracts the ID using @PathParam and then runs getPostById in PostService to retrieve the post data. If the post was located, it returned the post information with a 200 OK status; otherwise, it returned a 404 Not located error.

-> The @remove function with the /{id} path handles HTTP DELETE requests to remove a particular post based on its ID. It uses @PathParam to get the ID and then calls PostService's deletePostById method to remove the post. A successful operation provides a 204 No Content status, whereas an error generates a 500 Internal Server Error.

-> The @GET function, with the /user/{userId} path, gets all postings by a single user. It utilizes @PathParam to extract the userId and then calls getPostsByUserId in PostService to get the user's posts. If successful, it returns the posts with a 200 OK status; otherwise, a 500 Internal Server Error status is returned in the event of an error.

5) FollowController

-> The FollowController class has a constructor that initializes the FollowService, which handles the business logic of following and unfollowing users. This configuration guarantees that the controller can perform the following activities while also managing errors such as ClassNotFoundException and SQLException for stable operation.

-> The @POST function accepts HTTP POST requests for the /follows endpoint. It receives a FollowDTO object with followerId and followedId and uses FollowService's followUser method to follow the chosen user. A 201 Created status is provided upon success, whereas a 500 Internal Server Error is issued in the event of a SQLException.

-> The @DELETE function, using the /{followerId}/{followedId} path, handles HTTP DELETE requests to unfollow a person. It uses @PathParam to retrieve the followerId and followedId, and then runs unfollowUser in FollowService to end the relationship. A successful transaction provides a 204 No Content status, but any error generates a 500 Internal Server Error.

-> The @GET function, with the /followers/{userId} route, fetches a user's followers. It extracts the userId using @PathParam and then calls getFollowers from FollowService to retrieve a list of follower IDs. If successful, the procedure returns the list with a 200 OK status, and if a SQLException occurs, it returns a 500 Internal Server Error.

-> The @GET function, using the /following/{userId} URL, returns the users that a particular user follows. It retrieves userId using @PathParam and runs getFollowing in FollowService to obtain a list of following user IDs. If successful, the list is delivered with a status of 200 OK; otherwise, any exceptions result in a 500 Internal Server Error.

2- Implications after testing

After fully testing all our application's core features, we are satisfied that each component works as intended. Here's a summary of what we tried and the results:

- a) The UserPostLikeController efficiently handles the like functionality on posts. It employs a service to handle likes and dislikes, ensuring that tasks are completed correctly and problems are handled effectively. We validated that

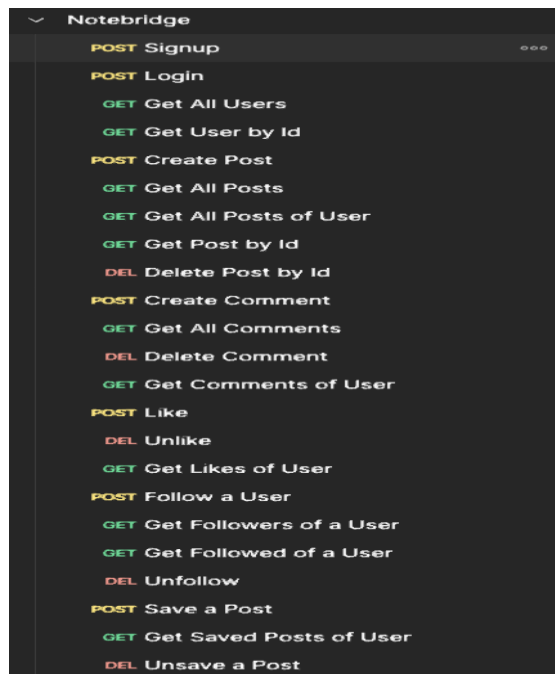
users may like and dislike postings without difficulty, and that the system handles faults such as database connection difficulties correctly.

- b) The `UserPostCommentController` manages comments. It lets users contribute, view, and remove comments on posts. Our testing revealed that all of these functionalities operate efficiently, and that the controller appropriately handles faults such as database issues.
- c) The `UserController` handles user-related tasks like as registering up, logging in, and retrieving user information. We tested these functionalities and discovered that users can register, log in, and access their data without issue. The controller also handles faults to ensure system stability and security.
- d) The `PostController` handles all post-related operations, including creation, deletion, and viewing. Our testing revealed that users may simply create and remove new posts, as well as view posts by ID or user. To ensure that consumers have a seamless experience, the controller effectively manages these activities and mistakes.
- e) Finally, the `FollowController` handles the following and unfollowing of users. It allows users to follow, unfollow, and see who is following whom. Our tests revealed that these functions perform properly, and the controller appropriately handles any potential issues.

As a result, our testing revealed that all of the major features—likes, comments, user administration, posts, and following—function properly and as planned. The program is dependable and user-friendly, and it handles mistakes in such a way that users have a consistent and good experience. Our extensive testing procedure ensured that the application works properly and is suitable for usage.

BLACK BOX TESTING:

In every step of coding, we used black box testing via Postman to ensure endpoints are working correctly, and to ensure appropriate response is given in case of invalid input etc. Below are the 23 endpoints currently implemented so far, each tested with multiple input values.



User Story Testing

The endpoints explained above are created according to user stories. The completed user stories are as follows:

- As a user, I want to be able to sign up using my email so that I can create a personal account
 - o POST http://localhost:8080/notebridge5_war/api/users/signup
 - o Consumes JSON in request body for user details while signing up
- As a registered user, I want to log in to my account using my credentials
 - o POST http://localhost:8080/notebridge5_war/api/users/login
 - o Consumes JSON in request body to login with email and password
- As a user, I want to create posts with details so that other users can interact with it
 - o POST http://localhost:8080/notebridge5_war/api/posts
 - o Consumes JSON in request body to classify posts
- As a user, I want to delete my posts
 - o DELETE http://localhost:8080/notebridge5_war/api/posts/{id}
 - o Consumes id from path parameter
- As a user, I want to classify my posts while creating so other users can find the post under a section with details
 - o POST http://localhost:8080/notebridge5_war/api/posts
 - o CreatePostDTO has multiple attributes that classifies and details the post
- As a user, I want to like/ comment on posts to interact with the community

- POST http://localhost:8080/notebridge5_war/api/likes
- Consumes JSON in request body to find userid and postid
- POST http://localhost:8080/notebridge5_war/api/comments
- Consumes JSON in request body to find userid and postid and comment text
- As a user, I want to save posts so that I can interact/see them later
 - POST http://localhost:8080/notebridge5_war/api/saves
 - Consumes JSON in request body to find userid and postid
- As a user, I want to see posts of other users in my home feed
 - GET http://localhost:8080/notebridge5_war/api/posts
- As a user, I want to see my saved/liked/commented posts from my profile page so that I can see/ interact with them easily -> Users should have quick access to their interactions, allowing them to revisit and further engage with posts they found interesting.
 - GET http://localhost:8080/notebridge5_war/api/likes/{userId}
 - GET http://localhost:8080/notebridge5_war/api/comments/user/{userId}
 - GET http://localhost:8080/notebridge5_war/api/saves/{userId}

Frontend done but endpoints still on test phase:

- As a user, I want to be able to search for posts so that I can find relevant content.
- As a user, I want to be able to edit my posts so that I can change them after posting
- As a user, I want to update my name, nickname, email address and description, so that my profile information is accurate
- As a user, I want to be able to upload and change my profile picture so that I can personalize my profile

UNIT TESTING:

Unit tests (codes are under test folder in src) for all of the services that carry out the business logic are implemented, and currently all the functionalities pass the tests without any errors. Below are the results of all the tests.


```
✓ UserPostSaveServiceTest (di5.serv 938 ms)
  ✓ testSavePost() 900 ms
  ✓ testUnsavePost() 14 ms
  ✓ testGetSavedPostsByUserId() 24 ms
Tests passed: 3 of 3 tests – 938 ms
```

```
✓ UserServiceTest (di5.services 1 sec 856 ms)
  ✓ testGetAllUsers() 1 sec 263 ms
  ✓ testCreateUser() 171 ms
  ✓ testGetUserById() 93 ms
  ✓ testGetUserToken() 181 ms
  ✓ testGetUserTokenInvalidPasswc 148 ms

✓ Tests passed: 5 of 5 tests – 1 sec 856 ms

NOTEBRIDGE: Connecting to jdbc:postgresql:
NOTEBRIDGE: Database connection created
```

Unit tests for all of the controllers are implemented as well.

Some of the tests are implemented to check whether sql exceptions are correctly returned or not in addition to checking functionalities. If there is something against our checks/rules on the database occurs, we are expecting sql exceptions. The others check functionalities of the controllers.

```
✓ FollowControllerTest (di5.controlle 996 ms)
  ✓ testGetFollowingSQLException( 957 ms)
  ✓ testFollowUserSQLException() 9 ms
  ✓ testUnfollowUserSQLException() 6 ms
  ✓ testGetFollowersSQLException() 7 ms
  ✓ testUnfollowUser() 4 ms
  ✓ testFollowUser() 5 ms
  ✓ testGetFollowers() 6 ms
  ✓ testGetFollowing() 2 ms
✓ Tests passed: 8 of 8 tests – 996 ms

Java HotSpot(TM) 64-Bit Server
java.sql.SQLException Create bra
    at di5.services.FollowServ
    at di5.controller.FollowCo
    at di5.controller.FollowCo
    at java.base/java.util.Arr
    at java.base/java.util.Arr
```

```
✓ PostControllerTest (di5.controller) 979 ms
  ✓ testGetPostById() 944 ms
  ✓ testGetPostsByUserId() 7 ms
  ✓ testGetAllPosts() 4 ms
  ✓ testGetPostByIdNotFound() 4 ms
  ✓ testGetPostsByUserIdException() 6 ms
  ✓ testDeletePostByIdException() 3 ms
  ✓ testCreatePost() 8 ms
  ✓ testDeletePostById() 3 ms
✓ Tests passed: 8 of 8 tests – 979 ms
"C:\Program Files\Java\jdk-17\bin\
Java HotSpot(TM) 64-Bit Server VM
Process finished with exit code 0
```

✓ UserControllerTest (di5.controll 1 sec 21 ms)

✓ testLoginByEmailAndPassword(983 ms)

✓ testGetAllUsers()7 ms

✓ testLoginByEmailAndPasswordUn.8 ms

✓ testCreateUser()10 ms

✓ testDeleteUser()3 ms

✓ testGetUserById()4 ms

✓ testDeleteUserException()4 ms

✓ testGetUserByIdNotFound()2 ms

✓ Tests passed: 8 of 8 tests – 1 sec 21 ms

"C:\Program Files\Java\jdk-17\bin\j
Java HotSpot(TM) 64-Bit Server VM w
java.lang.RuntimeException Create bre
at di5.services.UserService.del
at di5.controller.UserControlle
at di5.controller.UserControlle
at java.base/java.util.ArrayLis
at java.base/java.util.ArrayLis

Process finished with exit code 0

✓ UserPostCommentControllerTe 1 sec 79 ms

✓ testGetCommentsByPostIdS 1 sec 40 ms

✓ testDeleteComment()9 ms

✓ testGetCommentedPostsByUserId 3 ms

✓ testGetCommentsByPostId()6 ms

✓ testCreateComment()9 ms

✓ testDeleteCommentSQLException 5 ms

✓ testGetCommentedPostsByUserId 3 ms

✓ testCreateCommentSQLException 4 ms

✓ Tests passed: 8 of 8 tests – 1 sec 79 ms

✓ UserPostLikeControllerTest (di5.cont 1 sec

✓ testGetLikedPostsByUserId()972 ms

✓ testLikePostSQLException()9 ms

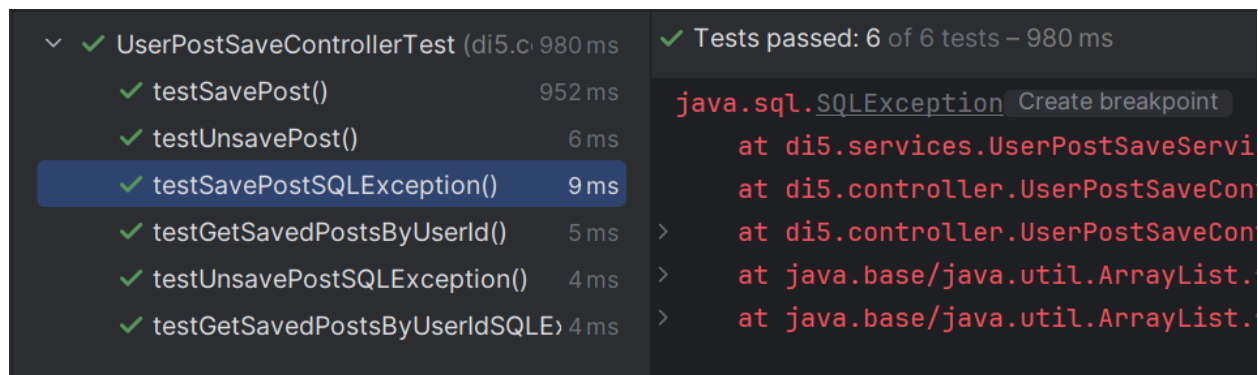
✓ testUnlikePostSQLException()6 ms

✓ testGetLikedPostsByUserIdSQLEx 5 ms

✓ testLikePost()4 ms

✓ testUnlikePost()4 ms

✓ Tests passed: 6 of 6 tests – 1 sec



```
✓ UserPostSaveControllerTest (di5.c 980 ms)
  ✓ testSavePost() 952 ms
  ✓ testUnsavePost() 6 ms
  ✓ testSavePostSQLException() 9 ms
  ✓ testGetSavedPostsByUserId() 5 ms
  ✓ testUnsavePostSQLException() 4 ms
  ✓ testGetSavedPostsByUserIdSQLException() 4 ms

✓ Tests passed: 6 of 6 tests – 980 ms

java.sql.SQLException Create breakpoint
  at di5.services.UserPostSaveServi
  at di5.controller.UserPostSaveCon
  at di5.controller.UserPostSaveCon
  at java.base/java.util.ArrayList.
  at java.base/java.util.ArrayList.
```

Integration Testing

Integration testing was carried out to ensure that the application's many components and services worked together seamlessly. The major goal was to test the interactions between different controllers, services, and the database. This involved verifying that controllers reliably execute service methods and that data flows properly between the controller and service levels. Special emphasis was placed on problem handling and response statuses for actions like as liking, commenting, and following articles. Database integration was also verified to ensure that data is correctly stored, retrieved, and edited, as well as how any problems such as `SQLException` are handled. The testing approach demonstrated that the system effectively manages inter-module communication and that the overall application remains stable over several operations. The integration testing relied heavily on ensuring database connectivity and resolving data-related errors correctly. We confirmed that the application components perform effectively by performing functionalities and interactions, resulting in a robust and dependable user experience.

Manual Testing

Manual testing was carried out to confirm the application's functioning from an end-user standpoint. This includes user interface testing to ensure that all components function properly, allowing users to engage with the program seamlessly. Signing up, logging in, posting posts, like, commenting, and following/unfollowing individuals have all been tested to verify they operate properly. Each feature underwent rigorous testing to ensure

that the program gives adequate feedback and handles user actions effectively. Functional testing entailed carefully verifying all essential features to ensure they worked as intended. The testing procedure revealed that the application is user-friendly and responsive, resulting in a consistent and favorable experience. By concentrating on real-life situations, we guaranteed that the application effectively matches user demands, with a special emphasis on graceful error handling and keeping a smooth flow of activities.