Xideral

Back End Academy

Final Project

Project By:

Kenneth Rodríguez García

Final Project: SmallTalk

Introduction

Project Overview

The main objective of this project is to develop the basic back end infrastructure for a microblogging social media platform called “**SmallTalk**”. This platform will enable users to create and interact with short-form text content.

Goals

The goal of this project is to provide a demonstration of all the concepts and technologies I learned during the Back End Academy. By leveraging these tools in a practical application, I hope to learn how they work and how to apply them in future projects.

Scope

The project will consist of a few key components to provide the necessary infrastructure for a minimum viable product.

- **User registration and authentication:** users should be able to create accounts, log in, and log out securely.
- **Post creation:** Users will be able to publish short text-based posts to the public feed.
- **Post feed:** All users will see the most recent posts published by other users in the platform.
- **Post liking:** Users can like other users’ posts, as well as their own, and the liked content should be persistent.
- **Real time updates:** Users will receive live updates when their content has been liked by other users.
- **Backup job:** A batch job to back up all of the system’s database should run on a scheduled basis.

Requirements

Functional Requirements

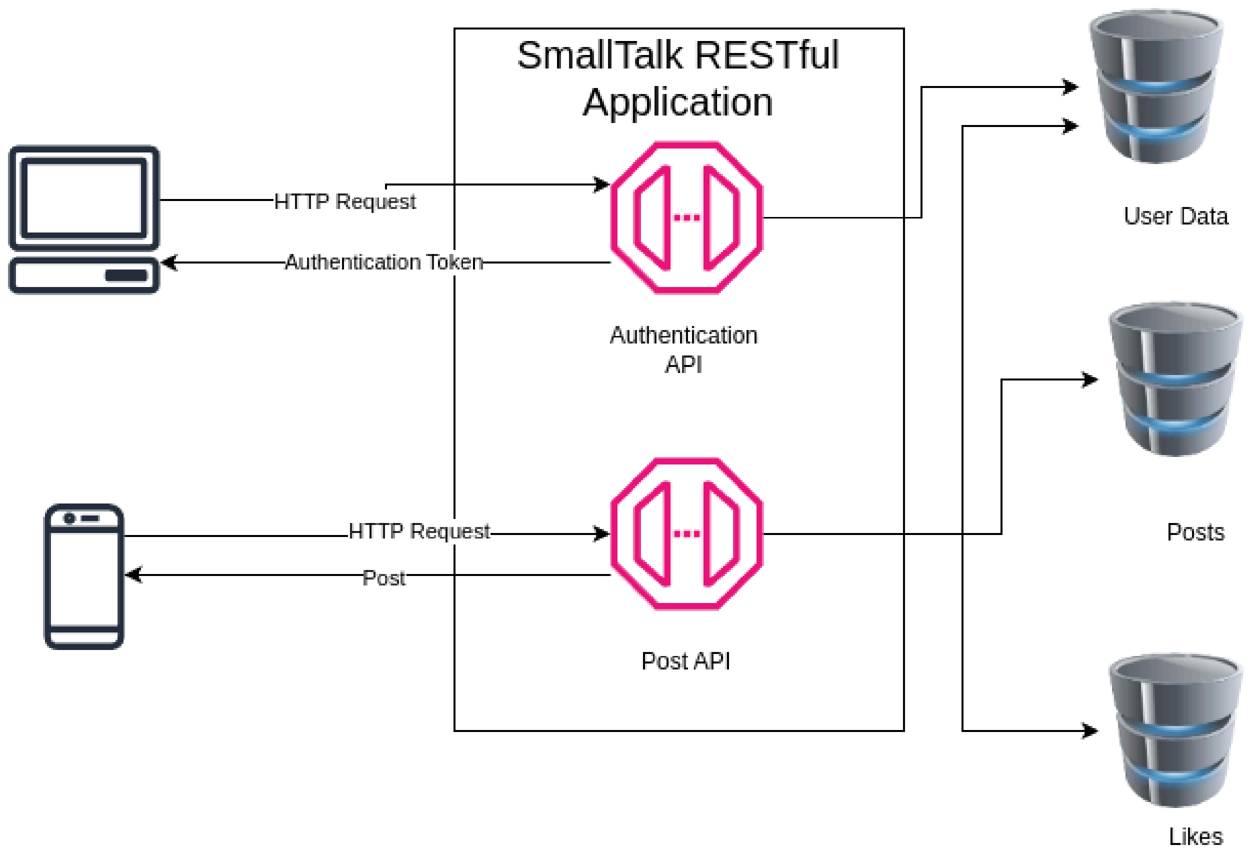
- User Registration and Authentication: Users should be able to create accounts, log in, and log out securely
- Publishing posts: Users should be able to publish posts for anyone to see.
- Viewing post feed: Users should see the latest posts in the public feed
- Sending Notifications: Users should receive a notification when one of their posts is liked.
- Back Up Job: The system should be back up its database once every day on a scheduled job.

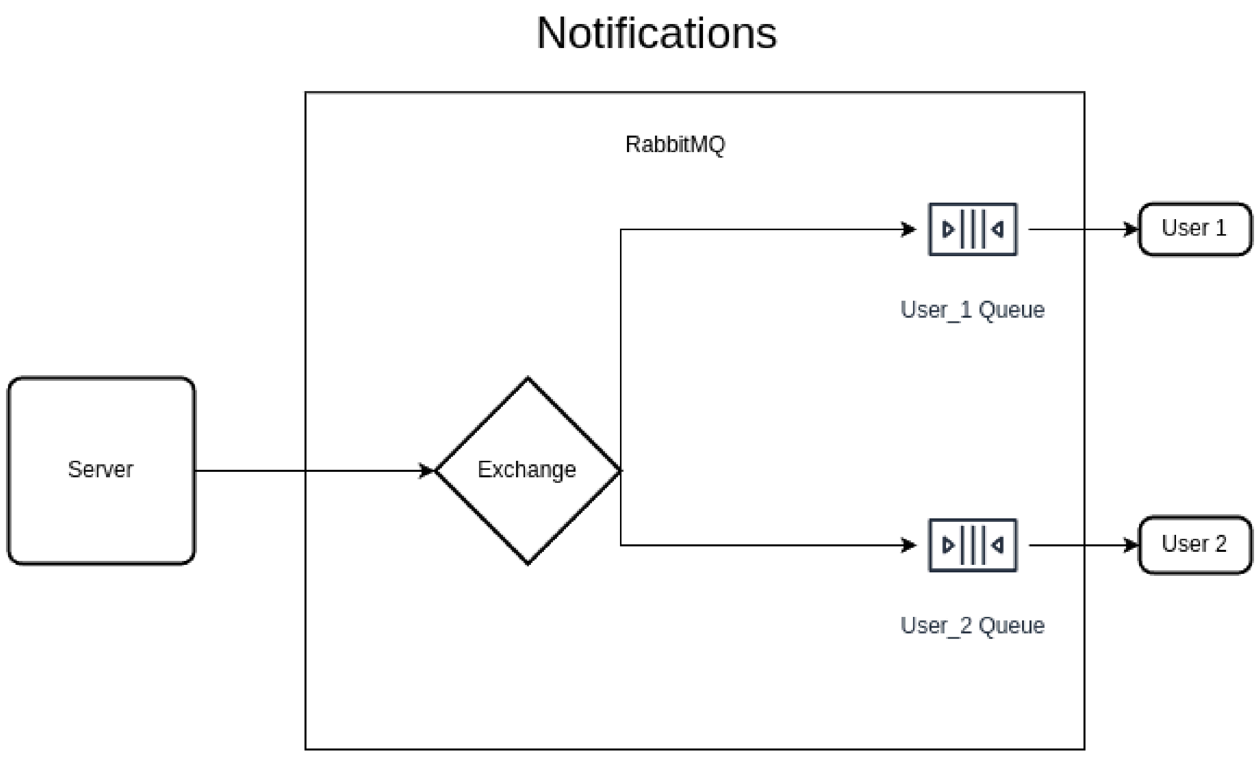
Non-Functional Requirements

- Security: User data should be stored securely preventing unauthorized access.

System Architecture

High-Level Diagram





Components

Back-End

The back-end infrastructure of the project is the basis of the entire service.

- **API Gateway**: The server provides a REST API entrypoint, which handles all incoming requests and routes them to the appropriate service.
- **User Service:** The user service handles requests for user registration and login. This service also provides the client with a user authentication token, which is necessary to access the other services provided by the application.
- **Post Service:** The post service manages post-related operations, such as publishing new posts or retrieving one or more posts to be viewed by users.
- **Like Service:** The like service updates the likes of a user created post, and triggers the notification service to notify the relevant users.
- **Notification Service:** The notification service sends a notification to a message broker which will route the message to its destination on the client side.
- **Back Up Job:** The backup job creates a daily backup of the existing database, to prevent data loss.

Database

The back-end uses a relational database to manage several necessary tables which contain the necessary user and post information.

- **User Data**: This table stores all user's username and their password hash
- **Post:** The "Post" table stores a posts content, as well as its original poster and date it was published.
- **Post Likes:** The "post-like" table manages which posts have been liked by various user's, and their current "liked" status (which is either true or false).

Message Broker

The platform uses RabbitMQ, a message broker, to send notifications to its various users. Each user has its own individual queue, which allows the system to provide personalized notifications.

Security

To secure user data, passwords aren't stored as plain text, but as password hashes. When a user logs in, their password is hashed using the SHA256 algorithm on the client side. The password hash is then compared with the stored password hash, and if there is a match the user will successfully log in.

A successful login returns a JWT to the user's client. This JWT is an authentication token which provides a session to the user. This token is necessary to validate the user's actions, such as publishing a post or liking content.

Front-End

A simple front-end client application was built with the Angular framework to provide a simple way of interacting with the API and message broker. The client also uses the Ionic framework to use its various UI components and improve the user experience.

Technology Stack

Back End

- Java 17 (version 17.0.12): An object-oriented programming language that runs on the Java Virtual Machine.
- Spring Framework (version 3.3.3): The Spring Framework provides a backbone to the project, which allows using its multiple components to build a full system.
    - Spring Boot: Embeds Tomcat to create a stand-alone application
    - Spring Web: Creates a RESTful web service
    - Spring Data: Allows for data access and persistence with relational databases.
    - Spring Batch: Processes large volumes of record in database, allowing for simple database backups.
    - Spring AMQP: Provides a template to interact with AMQP message brokers like RabbitMQ.
- MySQL (version 8.0.39): A relational SQL database that will store user data, posts and likes.
- Other Dependencies
    - MySQL Driver: the MySQL Connector J dependency provides the necessary drivers that allow interaction between the program and a MySQL database.
    - Lombok: This dependency provides useful annotations that reduce the amount of code in a project, providing setters and getters for attributes, as well as different constructors.
    - Java JWT: This dependency by Auth0 allows the creation of JSON Web Tokens to provide authentication and verify a user's identity.
- JUnit 5: A testing automation framework used to design unit tests.
- Mockito: A framework used to create mock objects and verify the behavior during unit testing.

Message Broker

- RabbitMQ (version 3.9.13): A message broker that receives messages in an exchange, and sends them to their destination queues to be consumed by another service.

Front End

- Angular: A TypeScript single-page web development framework with a diverse ecosystem.
- Ionic: A UI toolkit that provides HTML and CSS components to other web frameworks.

Implementation

Back-End

API Documentation

The server uses two main endpoints, “/users” and “/posts”, to handle authentication and post-related requests respectively.

Authentication Endpoints

Endpoint: “/users/register”

Method: POST

This method receives a User object, from a JSON object. The User has a username and password hash to be stored in the user_data table. This method returns a UserToken object, which includes the user’s ID, username, and a JWT for future authentication.

Endpoint: “/users/login”

Method: POST

This method receives a User object, from a JSON object, and verifies that the user’s username exists in the database and that the password hashes match. If they do, the method will return a UseToken object with the user’s ID, username, and JWT.

Post Endpoints

Endpoint: “/posts/{userId}/{token}”

Method: GET

This method receives two parameters from the route: a user’s ID and their authentication token (JWT). If the user’s identity and their JWT match, the method will return a list of the 10 latest posts, including the number of likes they have and whether they've been liked by the user. The list will consist of PostDTO objects, which aggregate data from both the Post table and PostLike table.

Endpoint: "/posts/upload"

Method: POST

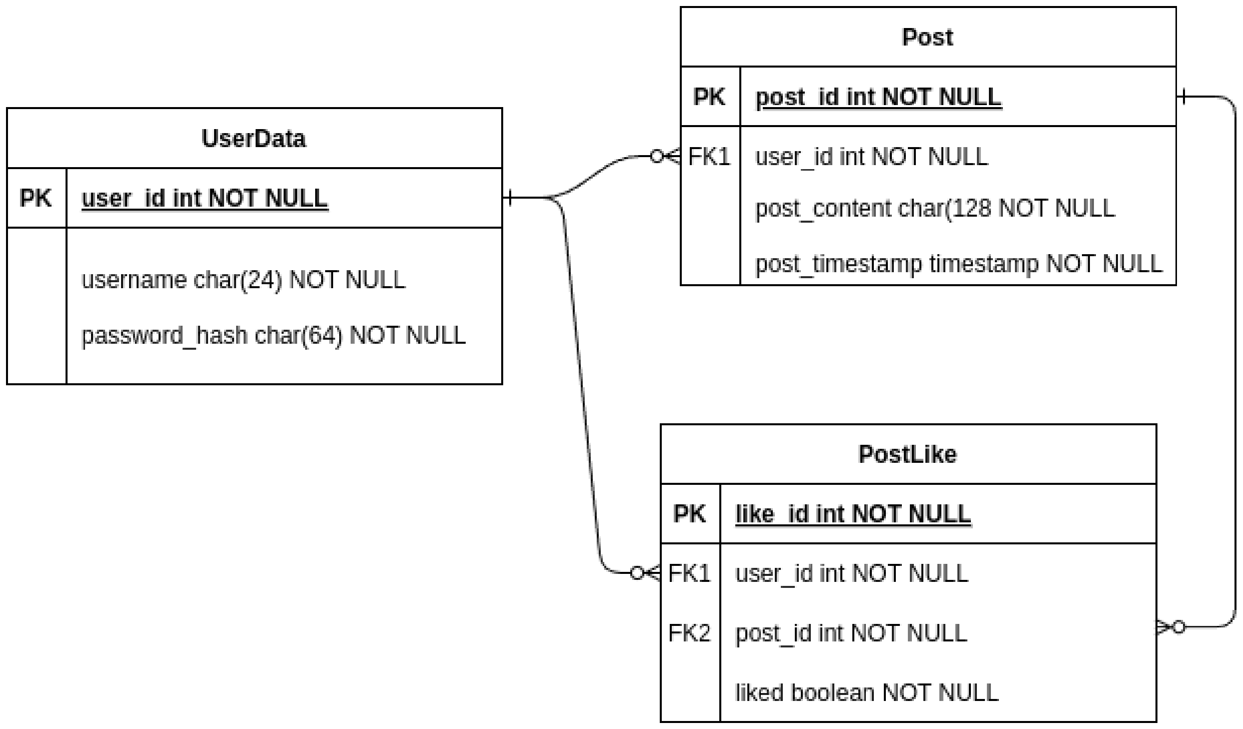
This method receives a Post object, and tries to save the post to the database. For this purpose, the user's JWT is required for authentication. If successful, the method will return a PostDTO object.

Endpoint: "/posts/like"

Method: POST

This method receives a Post object and updates the like status by the requesting user. If the post was liked, the method will send a notification to the original poster (only if the original poster and the user who liked the post are different users).

Database Schema



The application requires access to a database named "**SocialDB**", and this database should contain three different database tables: the User_Data table, the Post table, and the Post_Like table.

The creation of the database and its necessary tables can be achieved by executing the SQL script named “**create_db.sql**” found in the project’s files.

User_Data Table

This table stores the access credentials for all registered users in the platform. The table consists of only three different columns:

- UserID: An INT column, and primary key to the table. It identifies a specific user.
- Username: A VARCHAR(24) column. It represents each user’s username used to log into the application, and should be a unique identifier to avoid multiple users having the same username.
- Password_Hash: A VARCHAR(64) column. This column stores a password hash instead of a plain text password to provide more security to the users.

User Table	
Column	Data Type
UserID	INT, Primary Key
Username	VARCHAR(24), Unique
Password_Hash	VARCHAR(64)

Post Table

The Post table stores data for the text-based user generated content. Each post has its own unique id, as well as a reference to the user who posted it, among other attributes.

- PostId: An INT value, which is the primary key of each existing Post.
- UserId: An INT value and foreign key, which associates each Post to the user who published it.
- PostContent: A VARCHAR(128) value which represents the text contents of each post.
- PostTimestamp: A TIMESTAMP value that represents the timestamp of a post’s publication.

Post Table	
Column	Data Type
PostId	INT
UserId	INT, Foreign Key
PostContent	VARCHAR(128)
PostTimestamp	TIMESTAMP

Post_Like Table

The PostLike table determines whether a user has liked a given post or not. A new entry is only added when a user likes a post the first time. Each subsequent time the post is liked or disliked by the same user, the data entry is instead updated.

- LikeId: An INT value, and primary key for each entry.
- UserId: An INT value and foreign key to the User who liked (or this liked) the post.
- PostId: An INT value and foreign key to the liked post.
- Liked : A BOOLEAN value, which determines if a post is liked (or not) by the user.

Business Logic

The application implements 5 different services.

User Service

The User service handles all user registrations and authentications. When a user tries to register for the first time, this service will try to save the user's credentials (username and password hash) into the UserData table.

If a user is created successfully, the service will also create a message queue in RabbitMQ associated with the user. This queue is binded to the Users exchange using the new User's ID as the routing key. This message queue will be used to provide notifications to the user.

After the queue has been set up, the service will create a JSON Web Token, or JWT, to provide user authentication. This JWT will allow the user to securely post content and access their session in a browser of their choice, and has an expiration date of 30 days after the JWT’s creation. The token is associated with the username, and is signed using an HMAC with the SHA256 algorithm using a secret key.

Finally, the service will return a UserToken object, which contains the new user’s ID, username and the newly created JWT.

If an existing user is instead trying to log into their account, the user service will try to compare the given password hash with the password hash stored in the database. Similarly to the registering process, the service will verify the user has their own message queue (or create it if they don’t), generate a JWT, and return the UserToken object.

Post Service

The Post Service handles different post-related tasks, such as publishing and retrieving the user generated posts.

When a user tries to publish a new post to the feed, the service will use the user’s JWT to verify their identity and associate the post with the poster. If the user’s identity and their token do not match, the service will fail to publish the content to the feed.

Like Service

The Like Service updates a post’s liked value associated with a specific user, as well retrieving the number of likes a post has.

Notification Service

The Notification Service is an essential service for providing real-time notifications to users. This service uses a RabbitTemplate object (from the Spring AMQP component) to interact with the RabbitMQ message broker. When triggered, this service will send a message (composed of a JSON string) to the users’ exchange with the provided routing key.

Backup Job

The Backup Job is a scheduled job which runs every day at midnight. When triggered, the service will use the Spring Batch component to back up all of the available databases to .csv files, providing a way of rolling back the database in case of failure or data corruption.

The backup job consists of three steps, where each step is responsible for backing up one of the three different tables in the database.

Message Broker

To provide users with real time notifications, it is necessary to use a message-brokering system, which will allow users to receive updates without directly receiving the message from the application. The brokering system will instead act as a middle man, receiving the notifications and sending them to their destination.

RabbitMQ

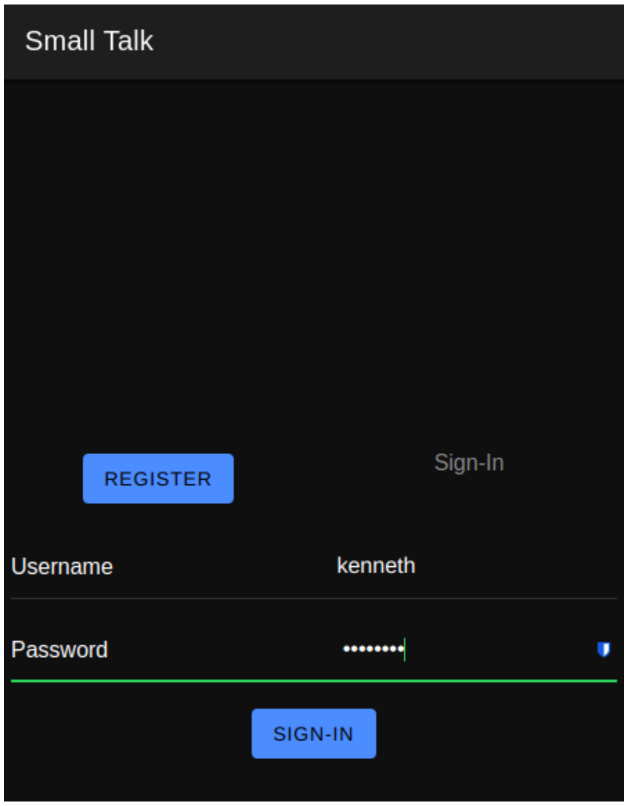
The RabbitMQ instance will receive messages from the server in an **Exchange**. The Exchange is in charge of receiving messages from a producer (our server, in this case), and sending them to the appropriate **Queue**. Each Queue is like a waiting list of messages, waiting to be consumed (received) by a consumer (a client).

As mentioned before, the Notification service sends updates to the Exchange named “**smalltalk.direct**”. This Exchange sends a copy of the message only to the queues where the routing key is an exact match. In our case, each queue is binded to the exchange by the ID of their respective user.

Front-End

An Angular application serves as a simple client that allows us to verify the behavior of our back-end server. The web app uses Angular and npm to manage its dependencies, but it also uses the Ionic framework to implement various UI components.

Users can choose to register a new account, or sign-in to an existing one.

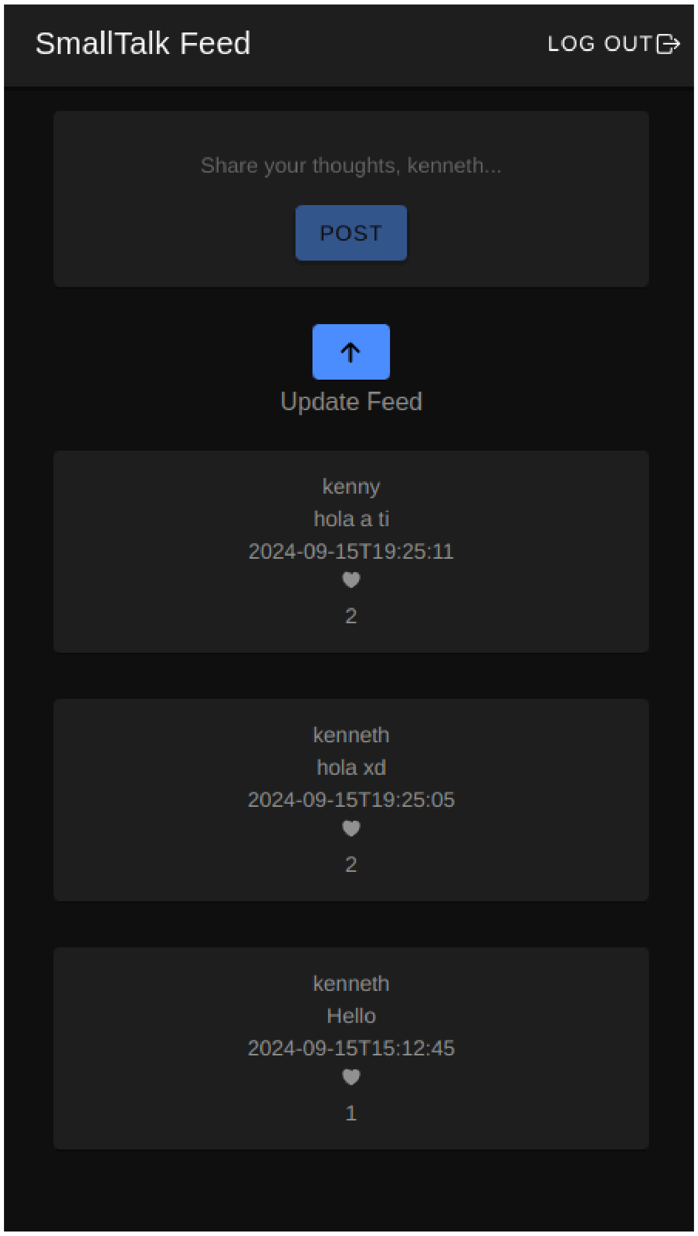


Username kenneth

Password ••••••••

Feed

After a successful log in, users will be welcomed by the SmallTalk Feed, which displays the latest posts published by all users in the platform.



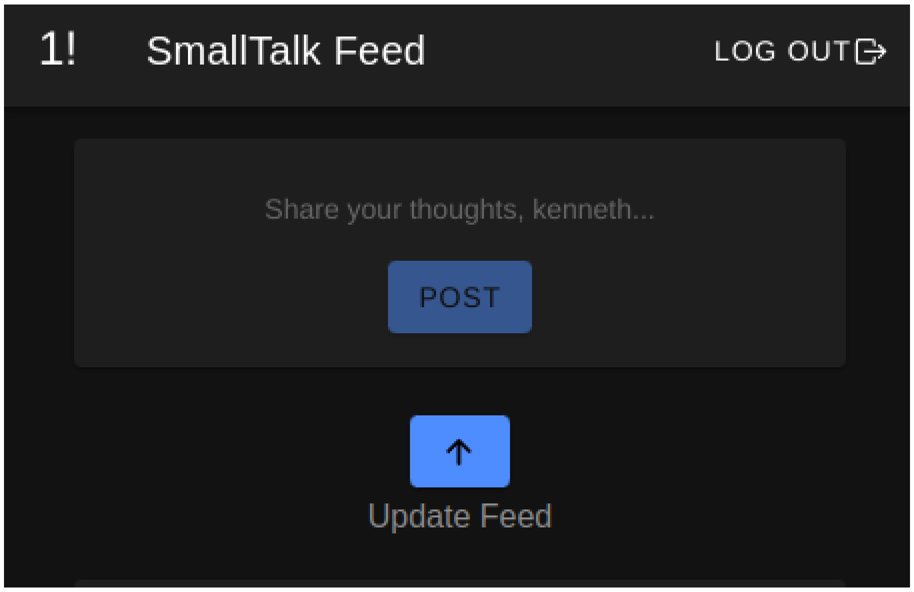
kenny
hola a ti
2024-09-15T19:25:11
2

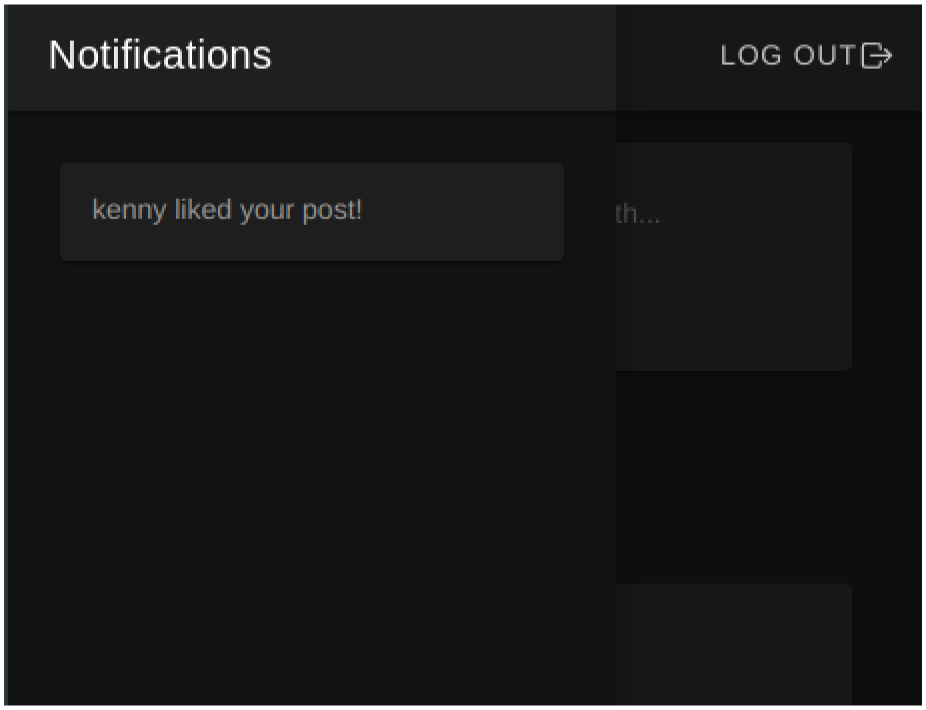
kenneth
hola xd
2024-09-15T19:25:05
2

kenneth
Hello
2024-09-15T15:12:45
1

Notifications

When one of the user's posts is liked by another user, they will receive a notification in the upper-left corner of the screen, and by clicking this button they will be able to see the notifications available to them.





kenny liked your post!

Testing

To ensure that everything works properly, it is necessary to set up a proper testing process that validates the correct execution of the application. To achieve this, Spring Boot provides the tools necessary to test the program.

We can leverage the JUnit 5 and Mockito frameworks to set up appropriate unit tests for the project. These dependencies will allow us to design specific tests that compare the expected behavior of the application against its actual execution.

Test Cases

Two main services will be tested: the User Service (which handles user registration and login), and the Post Service (which handles publishing and retrieving posts).

Using the Mockito framework, we can create mock services objects and define their behavior and expected results.

UserServiceTest

This class handles testing for the User Service.

Before each test, the *seUp()* method is executed, setting up a mock UserService object.

- testGetUserById: This test verifies that the user service can retrieve a user when given an ID assigned to an existing User.
- testUserNotFound: This test verified that the user service fails to retrieve a User object when given an ID that is not assigned to any User.
- testRegisterUser: This test registers that a User object is properly stored and registered in the system.

PostServiceTest

This class handles testing for the Post Service.

Similar to the user test, a setUp method is called to set up a mock PostService object. However, it also sets up a UserService object and a PostLikeService object.

- testGetPostById: This method validates that given a Post id, it will retrieve an existing Post, assigned to an existing User and retrieving the number of likes it has.

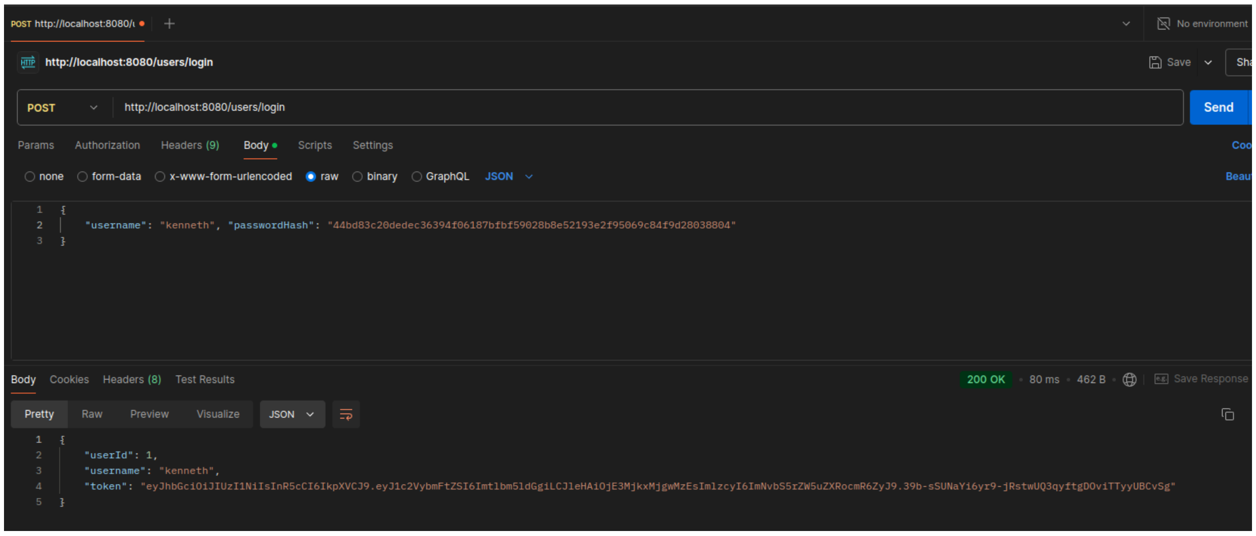
Testing Tools

As stated previously, the JUnit and Mockito frameworks are necessary tools to validate the testing process. However, they were not the only tools used.

Postman

The Postman platform allows us to test the API endpoints of our RESTful service. This software was used during the entire back-end development process, as it allowed testing of the program without the need of a client.

By providing easy access to the API, the feedback process was much faster, allowing for a shorter development process.



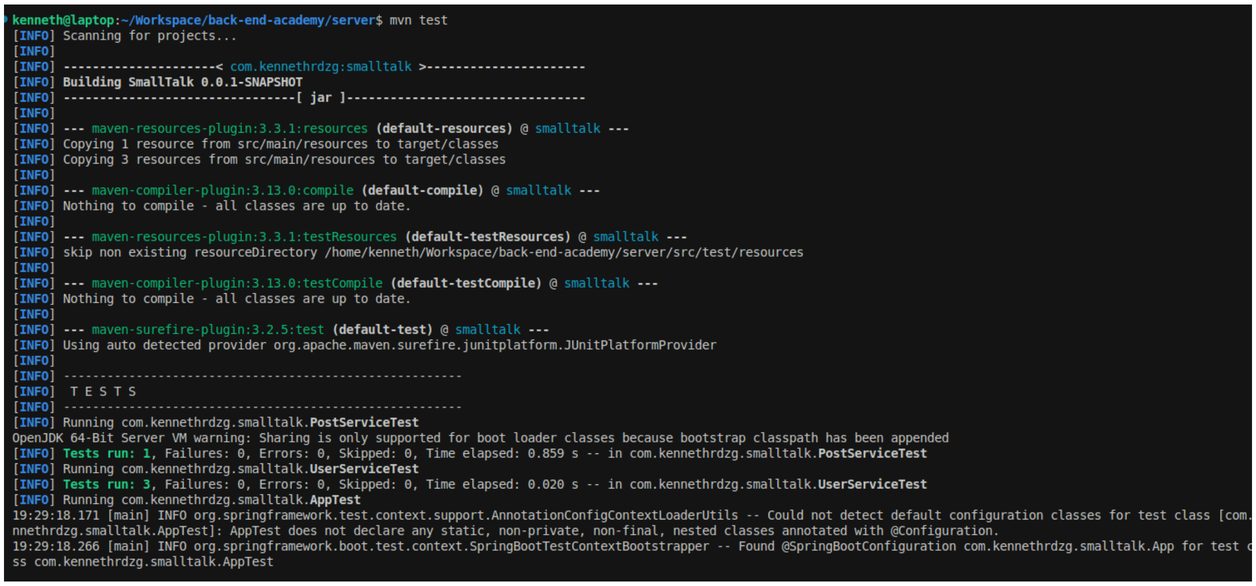
```
{
    "username": "kenneth", "passwordHash": "44bd83c20dedec36394f06187bfbf59028b8e52193e2f95069c84f9d28038804"
}
```

```
{
    "userId": 1,
    "username": "kenneth",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imtlbm5ldGgiLCJleHAiOjE3MjkxMjgwMzEsImlzcyI6ImNvbS5rZW5uZXRocmR6ZyJ9.39b-sSUNaYi6yr9-jRstwUQ3qyftgDOviTTyyUBCvSg"
}
```

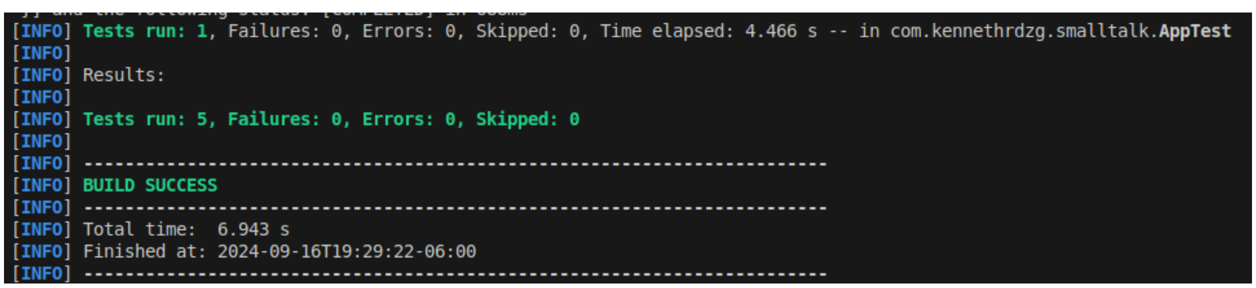
Maven

The Apache Maven tool not only managed the project's dependencies and automated the build process, but it also allowed for easier testing.

The Spring framework already provided the testing dependencies and necessary tools, so by simply executing the command "*mvn test*" on the command line, the entire testing process will be executed, as well as outputting the execution logs to the terminal.



```
kenneth@laptop:~/Workspace/back-end-academy/server$ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< com.kennethrdzg:smalltalk >---------------------
[INFO] Building SmallTalk 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]--------------------------------
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ smalltalk ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 3 resources from src/main/resources to target/classes
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:compile (default-compile) @ smalltalk ---
[INFO] Nothing to compile - all classes are up to date.
[INFO]
[INFO] --- maven-resources-plugin:3.3.1:testResources (default-testResources) @ smalltalk ---
[INFO] skip non existing resourceDirectory /home/kenneth/Workspace/back-end-academy/server/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.13.0:testCompile (default-testCompile) @ smalltalk ---
[INFO] Nothing to compile - all classes are up to date.
[INFO]
[INFO] --- maven-surefire-plugin:3.2.5:test (default-test) @ smalltalk ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.kennethrdzg.smalltalk.PostServiceTest
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.859 s -- in com.kennethrdzg.smalltalk.PostServiceTest
[INFO] Running com.kennethrdzg.smalltalk.UserServiceTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.020 s -- in com.kennethrdzg.smalltalk.UserServiceTest
[INFO] Running com.kennethrdzg.smalltalk.AppTest
19:29:18.171 [main] INFO org.springframework.test.context.support.AnnotationConfigContextLoaderUtils -- Could not detect default configuration classes for test class [com.
nnethrdzg.smalltalk.AppTest]: AppTest does not declare any static, non-private, non-final, nested classes annotated with @Configuration.
19:29:18.266 [main] INFO org.springframework.boot.test.context.SpringBootTestContextBootstrapper -- Found @SpringBootConfiguration com.kennethrdzg.smalltalk.App for test c
ss com.kennethrdzg.smalltalk.AppTest
```



```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.466 s -- in com.kennethrdzg.smalltalk.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.943 s
[INFO] Finished at: 2024-09-16T19:29:22-06:00
[INFO] ------------------------------------------------------------------------
```

Deployment

As a Java application, the web server can run on virtually any machine that runs the JVM with Java 17. However, the application needs some set up before proper installation and execution.

Deployment Environment

The application regularly accesses sensible systems, like the MySQL database or RabbitMQ. As such, it is bad practice to store the access credentials inside the codebase itself. Instead, during runtime the application will read environment variables which contain the necessary credentials to access all the required services.

To achieve this, a simple shell script which exports these variables is all that is required. In the current project set up, the environment variables should be in a shell script named “**env_variables.sh**”. This script will be ignored by the Git version control system, and it should not execute anything else.

The required environment variables are:

- **DATASOURCE_URL**: This is the URL that directs to the MySQL database. The format should be “**jdbc:mysql://hostname:port/SocialDB**”, where “hostname” is the hostname of the database and port is its access port (default value is 3306.
- **DATASOURCE_USERNAME**: This variable is the Username for accessing the MySQL database.
- **DATASOURCE_PASSWORD**: This variable is the password associated with the username variable mentioned above.
- **SERVER_PORT**: This variable represents the port where the Spring application will run, and it is necessary to access its endpoints.
- **APP_SECRET_KEY**: This secret key is by the REST service to sign and verify access tokens.
- **RABBITMQ_USER**: This variable provides the username to the RabbitMQ instance to publish messages.
- **RABBITMQ_PASSWORD**: This variable is the password associated with the RabbitMQ user mentioned above.

Deployment Process

As mentioned above, the “**env_variables.sh”** file should contain the necessary environment variables for the project’s correct execution. However, the script itself should not contain any other execution instructions. Instead, we use a second shell script to execute the project.

This shell script, named “**run.sh”**, is included in the project source files. The script should be executed in a shell from within the server’s directory. The script will call the environment variables script, and perform the cleaning, compiling, testing and execution process using Maven.

Alternatively, one may run the “**env_variables.sh”** script, and use the Maven command “**mvn package”** to package the project into a .jar file. However, it is necessary to note that if the project is executed using the .jar file, the application will still require the environment variables.

Future Enhancements

Potential Features

- User Following: Users could potentially follow other users and receive notifications when they post new content.
- Multimedia User Content: The platform could allow for content upload other than text, such as images, audio, or video.
- Password Salting: To increase security, the password salting technique could be implemented on top of the current password hashing.

Technology Considerations

- Containerization: By using tools like Docker and Kubernetes, multiple instances of the application could be easily deployed for improved scalability.
- Cloud Technologies: The project could eventually be moved from a local server to a virtual machine in a cloud provider.