**# Project: Spring Social media blog API**

**## Background**

Full-stack applications are typically concerned with both a front end, that displays information to the user and takes in input, and a backend, that manages persisted information.

This project will be a backend for a hypothetical social media app, where we must manage our users’ accounts as well as any messages that they submit to the application. However, the functionality for this project will leverage a popular web application framework for Java known as Spring. The Spring framework allows for automatic injection and configuration of many features, including data persitence, endpoints and conventional data manipulation logic (CRUD operations).

In our hypothetical micro-blogging or messaging app, any user should be able to see all of the messages posted to the site, or they can see the messages posted by a particular user. In either case, we require a backend which is able to deliver the data needed to display this information as well as process actions like logins, registrations, message creations, message updates, and message deletions.

**## Database Tables**

The following tables will be initialized in your project's built-in database upon startup using the configuration details in the application.properties file and the provided SQL script.

**### Account**

```

accountId integer primary key auto\_increment,

username varchar(255) not null unique,

password varchar(255)

```

**### Message**

```

messageId integer primary key auto\_increment,

postedBy integer,

messageText varchar(255),

timePostedEpoch long,

foreign key (postedBy) references Account(accountId)

```

**# Spring Technical Requirement**

**## Project must leverage the Spring Boot Framework**

Java classes have been provided, but your entire project MUST leverage the Spring framework.

In addition to functional test cases, "SpringTest" will verify that you have leveraged the Spring framework, Spring Boot, Spring MVC, and Spring Data.

SpringTest will verify the following

 - That you have, by any means, have a bean for the AccountService, MessageService, AccountRepository, MessageRepository, and SocialMediaController classes

 - That AccountRepository and MessageRepository are working JPARepositories based on their corresponding Account and Message entities

 - That your Spring Boot app leverages MVC by checking for Spring's default error message structure.

The app will already be a Spring Boot app with a valid application.properties and valid database entities at the start.

**# User Stories**

**## 1: Our API should be able to process new User registrations.**

As a user, I should be able to create a new Account on the endpoint POST localhost:8080/register. The body will contain a representation of a JSON Account, but will not contain an accountId.

- The registration will be successful if and only if the username is not blank, the password is at least 4 characters long, and an Account with that username does not already exist. If all these conditions are met, the response body should contain a JSON of the Account, including its accountId. The response status should be 200 OK, which is the default. The new account should be persisted to the database.

- If the registration is not successful due to a duplicate username, the response status should be 409. (Conflict)

- If the registration is not successful for some other reason, the response status should be 400. (Client error)

**## 2: Our API should be able to process User logins.**

As a user, I should be able to verify my login on the endpoint POST localhost:8080/login. The request body will contain a JSON representation of an Account.

- The login will be successful if and only if the username and password provided in the request body JSON match a real account existing on the database. If successful, the response body should contain a JSON of the account in the response body, including its accountId. The response status should be 200 OK, which is the default.

- If the login is not successful, the response status should be 401. (Unauthorized)

**## 3: Our API should be able to process the creation of new messages.**

As a user, I should be able to submit a new post on the endpoint POST localhost:8080/messages. The request body will contain a JSON representation of a message, which should be persisted to the database, but will not contain a messageId.

- The creation of the message will be successful if and only if the messageText is not blank, is not over 255 characters, and postedBy refers to a real, existing user. If successful, the response body should contain a JSON of the message, including its messageId. The response status should be 200, which is the default. The new message should be persisted to the database.

- If the creation of the message is not successful, the response status should be 400. (Client error)

**## 4: Our API should be able to retrieve all messages.**

As a user, I should be able to submit a GET request on the endpoint GET localhost:8080/messages.

- The response body should contain a JSON representation of a list containing all messages retrieved from the database. It is expected for the list to simply be empty if there are no messages. The response status should always be 200, which is the default.

**## 5: Our API should be able to retrieve a message by its ID.**

As a user, I should be able to submit a GET request on the endpoint GET localhost:8080/messages/{messageId}.

- The response body should contain a JSON representation of the message identified by the messageId. It is expected for the response body to simply be empty if there is no such message. The response status should always be 200, which is the default.

**## 6: Our API should be able to delete a message identified by a message ID.**

As a User, I should be able to submit a DELETE request on the endpoint DELETE localhost:8080/messages/{messageId}.

- The deletion of an existing message should remove an existing message from the database. If the message existed, the response body should contain the number of rows updated (1). The response status should be 200, which is the default.

- If the message did not exist, the response status should be 200, but the response body should be empty. This is because the DELETE verb is intended to be idempotent, ie, multiple calls to the DELETE endpoint should respond with the same type of response.

**## 7: Our API should be able to update a message text identified by a message ID.**

As a user, I should be able to submit a PATCH request on the endpoint PATCH localhost:8080/messages/{messageId}. The request body should contain a new messageText values to replace the message identified by messageId. The request body can not be guaranteed to contain any other information.

- The update of a message should be successful if and only if the message id already exists and the new messageText is not blank and is not over 255 characters. If the update is successful, the response body should contain the number of rows updated (1), and the response status should be 200, which is the default. The message existing on the database should have the updated messageText.

- If the update of the message is not successful for any reason, the response status should be 400. (Client error)

**## 8: Our API should be able to retrieve all messages written by a particular user.**

As a user, I should be able to submit a GET request on the endpoint GET localhost:8080/accounts/{accountId}/messages.

- The response body should contain a JSON representation of a list containing all messages posted by a particular user, which is retrieved from the database. It is expected for the list to simply be empty if there are no messages. The response status should always be 200, which is the default.

**## 9: The Project utilizes the Spring Framework.**

- The project was created leveraging the spring framework, including dependency injection, autowire functionality and/or Spring annotations.