# Design Document: Java Spring API for makeshift-Reddit API-1.0.0

## Introduction

The goal of this project was to develop a Java Spring API that emulates some functionalities of Reddit. The API will allow users to create, update, and delete posts, upvote/downvote posts and comments, view posts by a specific user, and retrieve comments along with post statistics.

### Architecture Overview

#### Technologies Used

* Java Spring Boot for API development
* Spring Data JPA for database operations.
* MySQL as the database management system
* Database Schema
* Intellij IDEA

Users table: Stores user information including username, email, and password.

Posts table: Contains information about posts such as title, content, creator ID, and upvote/downvote counts.

Comments table: Stores comments associated with posts along with their upvote/downvote counts.

//ADD OTHER TABLES HERE

//

### Endpoints

#### User Endpoints

POST /register: Register a new user.

POST /login: Authenticate user

#### Post Endpoints

POST /posts: Create a new post.

PUT /posts/{postId}: Update an existing post.

DELETE /posts/{postId}: Delete a post.

GET /posts/{postId}/comments: Get comments for a specific post.

#### Vote Endpoints

POST /posts/{postId}/upvote: Upvote a post.

POST /posts/{postId}/downvote: Downvote a post.

POST /comments/{commentId}/upvote: Upvote a comment.

POST /comments/{commentId}/downvote: Downvote a comment.

#### GET RESOURCE Endpoints

GET /users/{userId}/upvoted-posts: Get posts upvoted by a user.

GET /users/{userId}/downvoted-posts: Get posts downvoted by a user.

GET /users/{username}/posts: Get posts created by a specific user.

Detailed Implementation

## Service Layer with JPA

#### Post Service

Create Post: When a user submits a post creation request, the Post Service receives the necessary data (title, content, creator ID).

It utilises the PostRepository (an interface extending JpaRepository<Post, Long>) to persist the post data into the database.

The service method orchestrates the interaction between the controller and the repository, ensuring the creation of the post.

Update Post: Upon receiving an update request, the Post Service fetches the existing post entity from the repository using its ID.

It then updates the post attributes with the new data and saves the changes using the PostRepository.

Delete Post: The Post Service retrieves the post entity by its ID and instructs the PostRepository to remove it from the database.

#### Comment Service

Add Comment: Similar to creating a post, adding a comment involves receiving comment data (content, post ID, creator ID) from the controller.

The Comment Service interacts with the CommentRepository to persist the comment entity associated with the corresponding post.

#### Vote Service

Upvote/Downvote Post: When a user upvotes or downvotes a post, the Vote Service retrieves the post entity by its ID.

It updates the post's upvote/downvote count accordingly and persists the changes through the PostRepository.

Upvote/Downvote Comment: The Vote Service follows a similar process for voting on comments, interacting with the CommentRepository to update comment entities.

#### GET RESOURCE Service

Retrieve User's Upvoted/Downvoted Posts: Upon receiving a request to retrieve a user's upvoted or downvoted posts, the GET RESOURCE Service accesses the UserRepository to fetch the user entity.

It then retrieves the relevant posts based on the user's voting history from the PostRepository.

Retrieve Posts by Username: For fetching posts created by a specific user, the GET RESOURCE Service communicates with the UserRepository to find the user entity by username.

It then retrieves the corresponding posts from the PostRepository based on the user's ID.

#### Post Management

Users can create, update, and delete posts.

Posts have attributes such as title, content, creator ID, upvote count, and downvote count.

Updating or deleting a post will require authentication to ensure the user has the necessary permissions.

#### Comment Management

Users can comment on posts.

Comments are associated with a post and have attributes such as content, creator IDupvote count, and downvote count.

#### Voting System

Users can upvote or downvote posts and comments.

Each user can only vote once on a post or comment.

Voting will update the respective upvote/downvote counts.

#### GET RESOURCES

Users can retrieve posts they have upvoted or downvoted.

Users can view posts created by a specific user by providing their username.

Viewing a post will display its comments along with upvote and downvote counts.

### Full Authentication with Hashing IDs

#### Authentication

Full authentication involves securely verifying the identity of users before granting access to the API endpoints.

Upon registration or login , users' passwords are hashed using a secure hashing algorithm such as bcrypt before storing them in the database.

When users attempt to log in, the stored hashed password is compared with the hash of the provided password for authentication.

Successful authentication results in, providing access to authenticated endpoints.

### Hashing IDs for Security

To add an extra layer of security to the database, IDs (e.g., user IDs, post IDs) can be hashed before storing them.

Hashing IDs ensures that sensitive information like user identifiers cannot be easily inferred from the database.

Additionally, it prevents enumeration attacks where attackers attempt to guess valid IDs by incrementing or decrementing integers.

The hashing algorithm used enabled cryptographically secure and is irreversible, ensuring that IDs cannot be decrypted back to their original values.

#### Implementation

During user registration:

This identifier is hashed using a secure hashing algorithm and stored in the database alongside other user information.

When GET RESOURCE or accessing resources associated with a specific user, the hashed ID is used to retrieve the corresponding records/data.

Similarly, for posts and comments, unique identifiers are hashed before storage and used for referencing in the database.

### Quality Assurance

#### Cleanliness

Ensured that code was well-structured, following best practices and industry standards.

Proper naming conventions was followed to enhance readability.

#### Readability

Ensure that code was commented where necessary to explain complex logic or algorithms.

Consistent indentation and formatting was maintained throughout the codebase.

#### Optimization

Database queries were optimized to ensure efficient data retrieval and manipulation on repositories interfaces.

#### Scalability

The API was designed with scalability in mind through loose coupling , allowing for horizontal scaling by deploying multiple instances behind a load balancer.

Database design was consider scalability requirements to handle increasing data volumes.

#### Design Accuracy

The API adhered to the specified requirements, providing the necessary functionality to meet user needs.

Design decisions were justified based on factors such as performance, security, and maintainability.

### Design Patterns

* Builder Pattern
* Factory Method
* Singleton
* Proxy
* Facade
* Composite

Conclusion

This design document outlines the architecture and implementation details of a Java Spring API for a for makeshift-Reddit API-1.0.0 . By following best practices and considering factors such as cleanliness, readability, optimization, scalability, design accuracy, and design patterns, the makeshift-Reddit API-1.0.0 will be robust, efficient, and maintainable.