HU.25.1.2 – Java – Final Assignment

Problem Statement

Create a social network called ***SOCIO*** which emulates real life social networks and allows users to connect with each other.

Functional Requirements

- Create **Users** with varied roles in the platform.
    - A User is any person on the Socio platform.
    - Users can **register** with their email-id and a password.
    - Once registration is complete, users can **log in** with their credentials.
    - A user profile can be **public** or **private**. The user can toggle this setting by updating their profile.
    - There are two types of users in the platform:
        - **Regular USER** - public users who register using their credentials.
        - **Socio ADMIN** - people appointed by socio to moderate the platform. Their email ids must end with *@socio.com*.
    - A socio ADMIN should have all the privileges and rights of a regular user along with some additional privileges.
    - Only a socio ADMIN can add another socio ADMIN.
    - A socio ADMIN should be able to bulk import many users onto the platform using a *.csv* or a *.xlsx* file.

- Create **Posts** with different data types and metadata.
    - A post is any content posted on Socio platform.
    - A post can be a text with (or without) a file (ex: images, videos, documents, URL, etc.)
    - Metadata about a post should include the number of **likes** and **comments** on a post.
    - A post can be **reported** if it violates Socio policies.
    - A post can also be **shared** by another user.

- Interaction between Users
    - Users can **follow** other users.
    - Users can create Posts. If the user's profile is **private**, then their posts will be visible only to their followers (**this does not apply to SOCIO ADMINS**).
    - Users can **view** their own posts and posts of other users.
    - Users can **like** and **comment** in posts of other users.
    - Users can **share** posts of other users - this will create a new post on wall of the logged-in user, with the URL of the original user and the original post.
    - Users can **report** posts they don't like or disagree with a justification which would be up for moderation by ADMINs.

- Create **Groups** of Users
    - A User can create **groups**. One user can be part of multiple groups and there can be multiple groups with the same name but unique group ids.
    - Only the creator of a group can **add/remove** Users from a Group.
    - Only the members of a group can post in a group.
    - Groups can be **public** or **private**. The creator of the group can toggle this setting. If a group is **private**, then only the members of the group can see those posts in their feed (**this does not apply to Socio ADMINS**).
    - A logged-in user should be able to see all the groups he is part of as creator or member.
    - A logged-in user should be able to view details of such a group (i.e., which users are part of that group and who is the creator)

- Scheduled Interactions
    - A scheduler will run every day at 00:00 hrs. and create a **Happy Birthday from Socio** post for the User who has his/her birthday on that day.
    - This feature must be implemented using spring-batch and a Task Scheduler
        - A **reader** will read all the users.
        - A **processor** will check the users whose date of birth matches current day and flag them for the happy birthday post.
        - A **writer** will create the birthday post.

- Statistics and Moderation
    - An ADMIN should be able to see the following statistics:
        - All users ordered by follower count grouped by dates.
        - All posts ordered by number of likes and comments grouped by dates, users, and file type.
        - Reported posts grouped by dates, users, file type.
    - All the above reports should be exportable/downloadable in a .csv or .xlsx file.
    - An ADMIN should be able to view a report on a post and decide whether to keep the post up or take it down.
    - Accordingly, the report should be closed and either the post should be left as it is or deleted.

Non-functional requirements

- Application should have low latency and high throughput.
- API endpoints which return a large amount of data should be cacheable and paginated.
- The application and the data should be secured by authentication and role-based authorization.
- Security implementation must be stateless using JSON web tokens.
- The system should be scalable and maintainable.
- The system should implement exception handling to gracefully handle errors and provide meaningful error responses.
- The system should be properly unit-tested.

- Every functionality will be treated as fully completed only if it is accompanied by JUnits and Java Docs

Project Milestones

The assignment would be evaluated in an incremental manner based on five milestones. Each milestone below undertakes the functional/technical requirement as well as the weightage of the overall project

- **Milestone 1**: Setting Up workspace and Basic CRUD Operations and basic Spring security
    - **Weightage:** 15%
    - **Requirements**
        - Create a spring boot project.
        - Complete ER diagram and database setup.
        - Define Controllers, Service classes and DB classes to perform CRUD operations using JDBC
        - Integrate Spring Security for basic username-password authentication
        - Secure the APIs to ensure only authenticated users can access them.
        - Implement Swagger documentation.

- **Milestone 2**: JPA, Security, User registration, Authentication, and Logging
    - **Weightage**: 25%
    - **Requirements**
        - Implement ORM using JPA. Create Entities and DAO classes for CRUD operations
        - Integrate Spring Security for authentication and authorization using JWT.
        - Implement logging to track application events and errors.
        - APIs to throw unauthorized/unauthenticated error without valid passing JWT token.
        - Implement an API to register a new user
            - ✓ Should take user details, email and password as input and create a new user; credentials must be encrypted before storing in the database.
            - ✓ Should throw error if user is already registered.
            - ✓ Should be unauthorized/unauthenticated.

        - Implement an API to login a user
            - ✓ Should take username and password as input
            - ✓ Should return JSON web token with valid roles for the user
            - ✓ Should throw error if user is not present
            - ✓ should be unauthorized/unauthenticated
            - ✓ **If a user's password is expired (i.e., more than 30 days have passed since the password was last updated, then login should throw an error asking them to reset password)**

        - Implement an API to reset/change password of an existing user
            - ✓ API to change password of an existing user

- ✓ Should take username and password as input update credentials; credentials must be encrypted before storing
- ✓ Should throw error if user is already registered
- ✓ Should be unauthorized/unauthenticated

- Implement APIs to perform CRUD operations for user
  - ✓ Update user-details
  - ✓ Get ALL users with optional search params
  - ✓ Get ONE user using user id
  - ✓ Delete a user

- **Milestone 3**: Implement API to perform CRUD operations for user, posts, user-user interactions and user-groups
  - **Weightage**: 30%
  - **Requirements**

    - Implement APIs to perform CRUD operations for posts
      - ✓ Create a post by a user
      - ✓ Update a post by a user
      - ✓ Get all posts by a user with search params
      - ✓ Delete a post by a user

    - Implement APIs to perform user-user interactions
      - ✓ Implement API for a user to **follow/unfollow** another user
      - ✓ Implement API for a user to **comment** on their own or another user's post.
      - ✓ Implement API for a user to **like/unlike** their own or another user's post
      - ✓ Implement API for a user to **share a post** of another user. Sharing a post must create a duplicate of the original post with the URL of the original user and the original post.
      - ✓ Implement API for a user to **report** another user's post

    - Implement APIs to perform CRUD operations on user-groups
      - ✓ Implement an API to create a group. The user creating the group should be a member of the group by default.
      - ✓ Implement an API to add/remove other users from a group.
      - ✓ Implement APIs to create and update posts in a group.
      - ✓ Implement an API to see all groups with optional search params
      - ✓ Implement an API to see all user-specific groups (i.e., groups where the user is either a creator or a member)

- **Milestone 4:** Implement Admin operations, bulk-import, export, statistics and moderation (role-based authorization)
  - **Weightage**: 25%

- **Requirements**
  - Implement an API to add/update an ADMIN user
  - Implement an API to view all users ordered by follower count grouped by dates.
  - Implement an API to view all posts ordered by number of likes and comments and grouped by dates, users, file type and whether the post is linked to a group or not.
  - Implement an API to view all reported posts grouped by dates, users, file type.
  - Implement an API to view all groups sorted by the number of users and posts.
  - Implement an API for an ADMIN to act on a post which has been reported.
  - Implement an API to bulk-upload users using a .csv or a .xlsx file

- **Milestone 5**: Performance Optimization and Testing
  - **Weightage**: 10%
  - **Requirements**
    - Implement pagination for all the GET ALL APIs with searching functionality
    - Implement caching for all the GET ALL APIs where there is no pagination needed by design (e.g.: ADMIN reporting APIs). Implement a cache eviction policy to refresh cache when an ADD/UPDATE/DELETE operation is performed.
    - Implement Java Docs for all the classes implemented.
    - Optimize database queries for low latency and high throughput.
    - Ensure JUnit coverage of 85% or higher.
    - Implement logging for all the APIs and services.
    - Write one integration test on API of your choice to ensure reliability.

Milestone Evaluation

Milestone	Milestone Weightage	Task	Weightage
Milestone 1	15	Create a spring boot project.	3
		Complete ER diagram and database setup.	3
		Define Controllers, Service classes and DB classes to perform CRUD operations using JDBC	4
		Integrate Spring Security for basic username-password authentication	2
		Secure the APIs to ensure only authenticated users can access them.	2
		Implement Swagger documentation.	1
Milestone2	25	Implement ORM using JPA. Create Entities and DAO classes for CRUD operations	5
		Integrate Spring Security for authentication and authorization using JWT.	5
		Implement logging to track application events and errors.	1

		APIs to throw unauthorized/unauthenticated error without valid passing JWT token.	1
		Implement an API to register a new user	3
		Implement an API to login a user	3
		Implement an API to reset/change password of an existing user	3
		Implement APIs to perform CRUD operations for user	4
Milestone 3	30	Implement APIs to perform CRUD operations for posts	8
		Implement APIs to perform user-user interactions	8
		Implement APIs to perform CRUD operations on user-groups	14
Milestone 4	25	Implement an API to add/update an ADMIN user	5
		Implement an API to view all users ordered by follower count grouped by dates.	3
		Implement an API to view all posts ordered by number of likes and comments and grouped by dates, users, file type and whether the post is linked to a group or not.	3
		Implement an API to view all reported posts grouped by dates, users, file type.	3
		Implement an API to view all groups sorted by the number of users and posts.	3
		Implement an API for an ADMIN to act on a post which has been reported.	3
		Implement an API to bulk-upload users using a .csv or a .xlsx file	5
Milestone 5	10	Implement pagination for all the GET ALL APIs with searching functionality	2
		Implement caching for all the GET ALL APIs where there is no pagination needed by design (e.g.: ADMIN reporting APIs). Implement a cache eviction policy to refresh cache when an ADD/UPDATE/DELETE operation is performed.	2
		Implement Java Docs for all the classes implemented.	1
		Optimize database queries for low latency and high throughput.	1
		Ensure JUnit coverage of 85% or higher.	2
		Implement logging for **all** the APIs and services.	1
		Write one integration test on API of your choice to ensure reliability.	1