Implementation paper

Task Description

Objective	Task Name	Description
Development(Backend)	Perform CRUD Operation	1. The project should have a process of user registration, updation, deletion, and retrieval.
		2. The project should consider the unique username even after the deletion of one user the same should not be used.
		3. After registration a password should be generated (need not implement the sms part, return the password in response). This password should be directly linked with the user name. i.e. username can be retrieved via password. Ex. userName: ANURAG, password: 114211817 Note: This password mapping should reflect the strong encryption, logic behind the encryption should have a strong base. The retrieval should not be ambiguous (like the given example).
		4.Log Level and effectiveness of logs.

User Management Application

INDEX

1. Project Overview

1.1 Purpose

The **User Management Application** is a Spring Boot application that provides backend services for managing users. The system performs CRUD operations, manages user accounts securely through encrypted passwords, and ensures compliance with standards for user management.

1.2 Scope

The application is designed to handle user registration, updates, deletion, and retrieval, focusing on secure password management. It supports the following features:

- **User Registration**: Create new user accounts with unique username-associated encrypted passwords(which are generated automatically at time accounts creation)
- **User Management**: Update user information, retrieve user data, and handle account deletions.
- **Security**: Implement encryption for password security and ensure that deleted users are still accessible.

2. Project Structure

2.1 Directory Layout

.

sk@sk:~/Desktop/Shiva/Demo/TEN/user-management-application$ tree

```
.
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── keenable
│   │   │           └── user_magnagement_application
│   │   │               ├── controller
│   │   │               │   ├── ApiResponse.java
│   │   │               │   └── UserController.java
│   │   │               ├── model
│   │   │               │   └── UserEntity.java
│   │   │               ├── repository
│   │   │               │   └── UserRepo.java
│   │   │               ├── service
│   │   │               │   └── UserService.java
│   │   │               └── UserMagnagementApplication.java
```

```
│   │   └── resources
│   │       ├── application.properties
│   │
```

2.2 Key Components

- **`UserEntity.java`**: The entity class represents the `user` table in the database.
- **`UserService.java`**: Contains the business logic for user-related operations, including encryption and decryption of passwords.
- **`UserController.java`**: REST controller that exposes endpoints for managing users.
- **`UserRepository.java`**: JPA repository interface for interacting with the database.
- **`application.properties`**: Configuration file for database and server settings.

3. Application Setup

3.1 Prerequisites

- **Java 17**: The project is built and tested with Java 17.
- **Maven 3.9.3**: For managing dependencies and building the project.

3.2 Configuration

The `application.properties` file configures the database, server port, and other environment-specific settings:

Properties

```
# application.properties
# Enable H2 console
spring.application.name=user_magnagement_application
spring.h2.console.enabled=true

# Database connection settings
spring.datasource.url=jdbc:h2:file:/home/sk/test;AUTO_SERVER=TRUE


#spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
#spring.h2.console.enabled=true
```

```
# JPA / Hibernate settings
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update

# Show SQL queries
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

4. Detailed Implementation

4.1 Model Layer (`UserEntity.java`)

The `UserEntity` class is annotated with `@Entity` and maps to the database user table. Key attributes include:

- **userName** (`String`): Unique identifier for the user.
- **emailId** (`String`): User's email address.
- **password** (`String`): Encrypted password.
- **role** (`String`): User's role within the application.
- **salary** (`Double`): User's salary, stored as a decimal value.
- **phoneNumber** (`Long`): User's phoneNumber.
- **deleted** (`Boolean`): Indicates whether the user is marked as deleted.

Example code snippet:

```
package com.keenable.user_magnagement_application.model;
import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.annotation.JsonIgnore;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name="emp_user")
```

```
public class UserEntity {

   @Column(name="userName")
   @Id
   private String userName;
   private String emailId;
   private String password;
   private String role;
   @JsonFormat(shape = JsonFormat.Shape.NUMBER, pattern =
"0.00##//#endregion") //for Ignore E
   private Double salary;
   private Long phoneNumber;
   @JsonIgnore
   private Boolean deleted =false;


}
```

4.2 Service Layer (`UserService.java`)

4.2.1 Password Encryption and Decryption

The `UserService` class includes methods for password encryption and decryption. A simple Caesar cipher shifts characters for encryption:

- **Encryption**: Shifts each character by a defined number of positions (e.g., `SHIFT = 4`).
- **Decryption**: Reverses the shift to retrieve the original password.

Example encryption method:

java code

```
private String encrypt(String input) {
       StringBuilder encrypted = new StringBuilder();
       for (char i : input.toCharArray()) {
           if (Character.isLetter(i)) {
               char base = Character.isLowerCase(i) ? 'a' : 'A';
               encrypted.append((char) ((i - base + SHIFT) % 26 +
base));
           } else if (Character.isDigit(i)) {
               encrypted.append((char) ((i - '0' + SHIFT) % 10 + '0'));
           } else {
               encrypted.append(i);
           }
```

```
        }
        return encrypted.toString();
    }

    // Decrypt the password
    private String decrypt(String input) {
        StringBuilder decrypted = new StringBuilder();
        for (char i: input.toCharArray()) {
            if (Character.isLetter(i)) {
                char base = Character.isLowerCase(i) ? 'a' : 'A';
                decrypted.append((char) ((i - base - SHIFT + 26) % 26 +
base));
            } else if (Character.isDigit(i)) {
                decrypted.append((char) ((i - '0' - SHIFT + 10) % 10 +
'0'));
            } else {
                decrypted.append(i);
            }
        }
        return decrypted.toString();
    }
```

4.2.2 CRUD Operations

The `UserService` class contains methods for managing users:

- **Create User**: create user by username with other data as emailid, role,phonenumber etc
- **Get User by Username**: Retrieves user details without the password.
- **Get all User** : Retrieves user details without the password.
- **Update User**: Allows updating user information.
- **Delete User by Username**: Marks a user by username as deleted without removing them from the database.
- **Deleteall User**: Marks all user as deleted without removing them from the database.
- **Get User by password**: Retrieves user details using the password.

Example create user method:

java code

```
public UserEntity createUser(UserEntity user) {
        String encryptedPassword = null;
        if (user.getUserName() != null) {
```

```
            encryptedPassword = encrypt(user.getUserName()); // Encrypt
username to create password
            user.setPassword(encryptedPassword); // Set the encrypted
password temporarily
        }
        UserEntity savedUser = userRepo.save(user);
        savedUser.setPassword(encryptedPassword); // Include the
encrypted password only in the registration response
        return savedUser;
    }
```

4.3 Controller Layer (`UserController.java`)

The `UserController` class defines RESTful endpoints for interacting with the user data. Some of the key endpoints include:

- **POST `/register/{username}`**: Registers a new user and returns the user account(details) with a generated (automatically)encrypted password.
- **GET `/retrieve`**: Retrieves all user details.
- **GET `/retrieve/{username}`**: Retrieves a user by username.
- **PUT `/update/{username}`**: Updates specific user information.
- **DELETE `/delete`**: Soft deletes all users by marking them as deleted.
- **DELETE `/delete/{username}`**: Soft deletes a user by marking them as deleted.
- **GET `/retrieve/password/{encryptedPassword}`**: Retrieves the username associated with an encrypted password.

Example controller method:

Java code

```
@PostMapping("/register/{username}")
public ResponseEntity<UserEntity> registerUser(@PathVariable String
username, @RequestBody UserEntity user) {
    user.setUserName(username);
    UserEntity createdUser = userService.createUser(user);
    return
ResponseEntity.status(HttpStatus.CREATED).body(createdUser);
}
```

4.4 Repository Layer (`UserRepository.java`)

The `UserRepository` interface extends `JpaRepository`, providing methods to query the database:

- **findByUserName**: Retrieves a user by their username.
- **findByDeletedFalse**: Retrieves all users that are not marked as deleted
- **findByUserNameAndDeletedFalse:** Retrieves a user by username if they are not marked as deleted.

Example repository interface:

java code

```java
@Repository
public interface UserRepo extends JpaRepository<UserEntity, String> {
   Optional<UserEntity> findByUserName(String userName );
   List<UserEntity> findByDeletedFalse();

   Optional<UserEntity> findByUserNameAndDeletedFalse(String username);
```

5. Deployment(Not Production level)

5.1 Building the Application

The application is built using Maven. Use the following command to build the project:

```
mvn clean install
```

5.2 Running the Application

Run the application using the following command:

```
mvn spring-boot:run
```

5.3 Accessing the Application

The application can be accessed at `http://localhost:8080`. Use tools like Postman or curl to interact with the API endpoints.