Object Oriented Programming

Final Project Report:

Binus University

Binus International

Student Information:

Name: Justin Hadinata

Student ID: 2702298236

Course Code: Comp6699001

Course Name: Object Oriented Programming

Class: L2AC

Lecturer: **Jude** Joseph Lamug Martinez, MCS

Type of Assignment: Final Project Report

A. Introduction

1. **Background**

   I've always been interested in social media, networking, and SQL. So, during this semester, I want to sharpen my coding and debugging skills, and decided to build and maintain a full stack website. I've always been interested in exploring the back-end part of a website, because so far in our semester the lecturer has only taught us about the front-end part such as JS, HTML, and CSS.

2. **Identified Problems**

   I find that it is quite hard for some people to make friends because making friends with strangers online can be quite daunting and difficult. I believe that people could make friends easier with fellow like-minded individuals, which can be in terms of a hobby, interest, favorite music, and more.

B. Project Specification

1. **Project Description**

   LoopPop is a social media web-application that is created primarily using Java. The design of the project is designed to be similar to MySpace and Reddit, as the project has user profile customization and a live forum comment section. The project has user authentication, and a powerful password encryption. The project allows users to interact with others, create their own accounts, customize their own profile such as their favorite hobby, and music, and also view other people's profiles. Users can first enter the Index page or the Landing page and navigate through the page to further learn more about the project's Information, features, and more.

If the user is interested, they then can make an account in the registration page, and then login in the login page. After a successful login / registration process, the user will be redirected to the main page, where most of the features that our website provided reside. In the main page, the user will gain access to many features, such as editing and customizing their profile, interacting with others by chatting in the comment section, and viewing other people's profiles!

All of the features utilizes SQL, as every relevant data such as the user's first name, favorite hobby, the content residing in the comment, etc are stored in a Database.

2. **Libraries / Modules / Technology / Programming Language**

All the libraries, technology, and modules that is used on this project other than Java:

- Thymeleaf
    - Java-based template engine for rendering HTML templates on the server side
- SpringBoot
    - Framework for creating the standalone, production-grade Spring-based applications
- SpringSecurity
    - Authentication and Authorization framework provided by spring for security.
- Jakarta
    - Java Platform used for building large-scale, distributed applications
- Maven
    - Build Automation tool and dependency management for Java projects
- HTML

- For the Frontend, markup language used for structuring web pages

- CSS

- For the Frontend, decorates the HTML

- JS

- Add interactivity and behavior to web pages.

- PostGreSQL

- To manage the database

- DBeaver

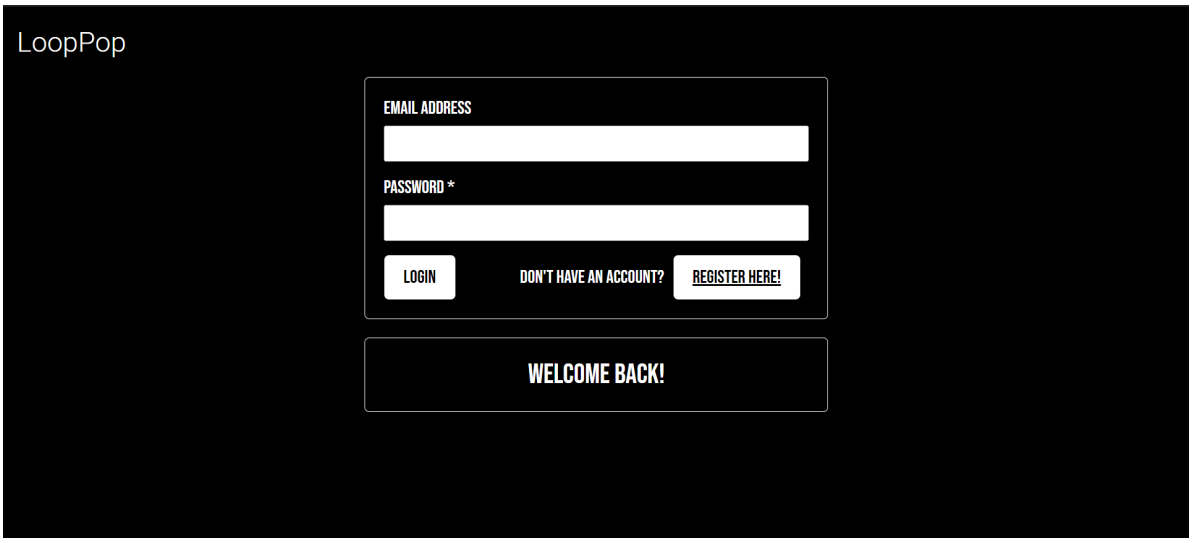
- To connect the IntelliJ with the PostgreSQL

3. **The website's feature overview**

Authentication Process:

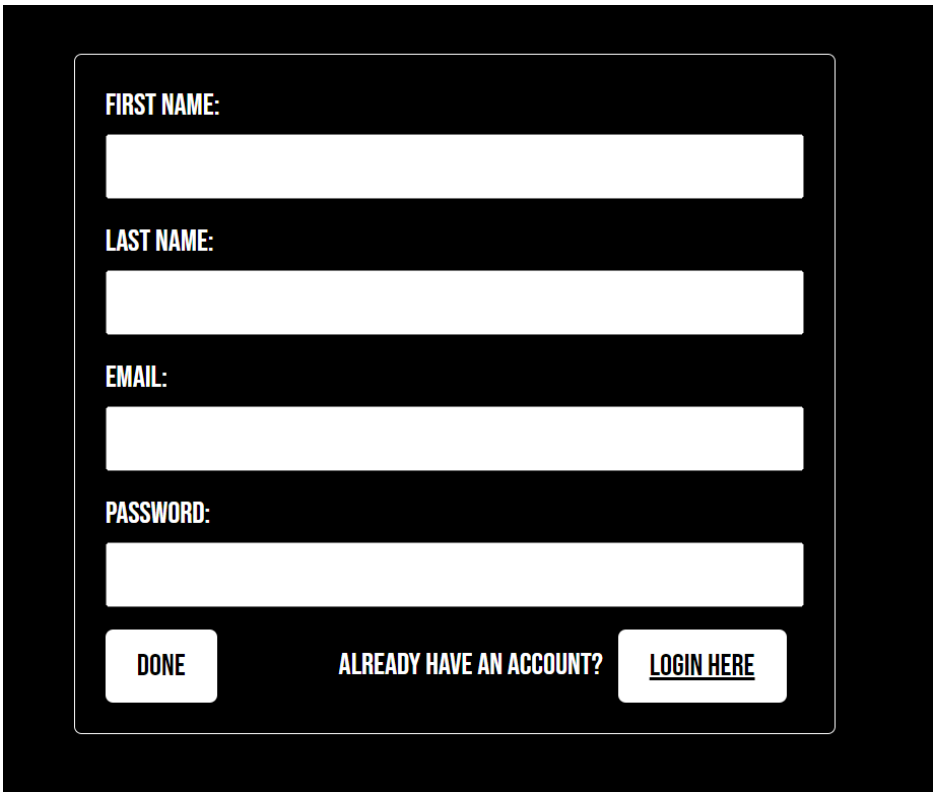
Login page.

User are able to login in the login page if they already have an account, after a successful login, it will redirect the user to the main page.



Registration Page

If the user haven't made an account yet, the user can create an account in the Registration Page.



Main Page:

This is the main page where most of the features reside. Users can Interact with others, comment, view other people's profile and edit their own profile !

4. **Algorithm & Code.**

The project have a few packages:

Comment, LoopPop_User, AppUser, Registration, and Security.

Before continuing on explaining the Code and Algorithm, I need to explain the File Packages purposes and connection:

AppUser

AppUser Package is responsible for the Authentication part of the web-application, it utilizes Spring security to handle the security. The database is stored in SQL.

LoopPop_User

LoopPop_User is responsible for the user's information, and allows them to customize this information to be shown to others and view others profiles.This entity will be used by the Comment entity, to show the commentator's identity.

Comment

Comment is response for the comment features, the live comment features works by storing the information about the comment (Content, tag, etc) in a SQL database.

Registration

Registration is responsible for handling most of the API requests such as Get request and post request, which is responsible for showing the HTML page and to take the information a user wants to give to the database (Post request for the user's profile customization). It is also responsible for signing up for the user for the authentication part.

Security

The Security handles like authentication logic, it have two parts:

- SecurityConfig
-PasswordEncoder

The password Encoder purpose is to hash the user's password, so even the project's developer (Me) cannot see the password.

SecurityConfig deals with the authentication logic, such as what /path can the user visit when not authenticated, and can when authenticated. It also used for CSRF Protection (Cross-site-request-forgery). I will starts explaining the code with the Security part.

The Code / Algorithm:

PasswordEncoder

```
package com.LoopPop.LoopPop.Security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
```

This part of the code shows that we imported springframework and spring security. Which is needed for this security part.

```
@Configuration
public class PasswordEncoder {

    no usages   Justin Hadinata
    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() { return new BCryptPasswordEncoder();
}
```

The @Configuration annotation indicates the file is part of the configuration class for the SpringBoot. @Configuration class is used to inject dependencies.

The @Bean annotation indicates the method or class is part of the bean class. @Bean annotation basically means that the class or method is managed by Spring Container. So the @Configuration annotation means that we’re injecting this @Bean class, the class is instantiated every time a company declares a dependency on the class.

SecurityConfiguration

```java
@Configuration
// Enable Spring Security's web security support
@EnableWebSecurity
public class SecurityConfiguration {

    // Define a bean for the security filter chain
    no usages   Justin Hadinata *
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .csrf(csrf -> csrf
                        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
                        .ignoringRequestMatchers( ...patterns: "/registration*", "/login*", "/api/v1/comments")
                )
```

The @Configuration and @Logic are the same as before, the @EnableWebSecurity is to enable spring security in spring application.
So basically this part of the code handles authorization rules and CSRF protection. In the .csrf, we're configuring the CSRF protection, and in the configuration we're disabling the CSRF Protection for some endpoints, such as "/registration*", "/login*/, and "/api/v1/comments".

```java
.authorizeRequests(authorize -> authorize
        .requestMatchers( ...patterns: "/registration*", "/login*", "/error",
                "/css/**", "/js/**", "/images/**", "/", "/api/v1/comments").permitAll()
        .anyRequest().authenticated()
)
```

This part of the code is to configure the authentication rules. In the .requestMatchers, we're basically saying that we're allowing unauthorized access to some of the endpoints, by permitting them. (Through the function .permitAll). And .anyRequest().authenticated() means that any other endpoints require authentication.

```
.formLogin(form -> form
        .loginPage("/login")
        .defaultSuccessUrl( defaultSuccessUrl: "/main",  alwaysUse: true)
        .usernameParameter("email")
        .permitAll()
)
.logout(logout -> logout
        .invalidateHttpSession(true)
        .clearAuthentication(true)
        .logoutRequestMatcher(new AntPathRequestMatcher( pattern: "/logout"))
);
```

This part of the code is used to configure the login form. It basically means that we want /login page to be our login page, if login is successful, we would redirect the user to the "/main" page, and we make sure that it will always be the case. We then specify the parameter name for the username which is the email, and the .permitAll basically means that all of the users are permitted or allowed to visit the login form. The .logout is to configure the logout functionality logic, it basically sets the logout request url, invalidates the HTTP session on logout, and clears the authentication information on logout.

```
return http.build();
```

Finally, we build the object and return the HTTPSecurity Object.

Registration:

RegistrationController

```
@Controller
public class RegistrationController {

    // Dependencies that will be injected
    2 usages
    private final RegistrationService registrationService;
    1 usage
    private final AppUserService appUserService;
    5 usages
    private final LoopPop_UserService loopPop_UserService;
```

The @Controller annotation is used to indicate that the class is part of the controller, which is responsible for the API request. This is basically the overview of the SpringBoot architecture:

API Controller Layer -> Business Service -> Data Link.
API Controller Layer -> Basically handles all of the API requests such as Get, Post, Delete, and Put method, which is also known as CRUD. This layer focuses on request parsing, and response request, and DID not have the logic to do so. That's what the Business Service does.

Business Service layer -> Basically handles all of the logic for the API Layer, API layer usually will inject Business Service as a dependency and use the methods provided by this layer. Business layer did not handle the API request. Business Layer however, did not hold the data, rather it performed requests such as CRUD to the data. The entity that holds the data is called the Data Link layer.

Data Link -> Data link didn't exactly hold the data literally, rather it interacted with the database.

Back to the code : As you can see from the code, we will use the business service layer from several packages, the registration itself, appuser, and the looppop_user.

```
no usages   Justin Hadinata *
public RegistrationController(RegistrationService registrationService,
                              AppUserService appUserService,
                              LoopPop_UserService loopPop_UserService) {
    this.registrationService = registrationService;
    this.appUserService = appUserService;
    this.loopPop_UserService = loopPop_UserService;
}
```

This code serves as the constructor for the dependency injection so we can use them.

```
// Endpoint to redirect the root URL to the index page
no usages   Justin Hadinata
@GetMapping("/")
public String defaultPage() { return "redirect:/index"; }

// Endpoint to serve the index page
no usages   Justin Hadinata
@GetMapping("/index")
public String index() { return "index"; }

// Endpoint to serve the login page
no usages   Justin Hadinata *
@GetMapping("/login")
public String login(Model model) {
    // Add a new RegistrationRequest object to the model
    model.addAttribute( attributeName: "user", new RegistrationRequest());
    return "login";
}
```

Now we're focusing on the API request.

@GetMapping -> Get method, which means the user requests to see an information and we provide it to them.

@PostMapping -> Post method, which means the user requests to give an information and we take it from them.

@PutMapping -> Put method, which means the user requests to configure or change an already existing information.

@DeleteMapping -> Delete method, which means the user requests to delete an information.

The @GetMapping("/), means that if the user visited this path (Example: http://localhost:8080/), it will redirect them to the index page. These

@GetMapping are basically an endpoint, waiting for the user's request. The @GetMapping ("index") to handle the index page, if the user visited the path, it will show the user the index page. The @GetMapping ("/login"), is also the same as before, to show the login page, but the Model is used to be passed onto Thymeleaf, basically give the information to the frontend so that the frontend can display information based on the backend.

```
// Endpoint to serve the main page
no usages   Justin Hadinata *
@GetMapping("/main")
public String mainPage(Model model, @AuthenticationPrincipal UserDetails userDetails) {
    if (userDetails == null) {
        // Redirect to the login page if the user is not authenticated
        return "redirect:/login";
    }

    String email = userDetails.getUsername();
    LoopPop_User existingUser = loopPop_UserService.findByEmail(email);

    if (existingUser != null) {
        // Add the user's first name and user object to the model
        model.addAttribute( attributeName: "firstname", existingUser.getName());
        model.addAttribute( attributeName: "loopPopUser", existingUser);
    }

    return "main";
}
```

This part of the code handles the Get Request for the /main. The @GetMapping works the same as before. The code basically will redirect to the login page if the user is not authenticated, which can be seen in the if statement. Then the code will get the user firstname and email by get .getUsername and .findByEmail which is a method that I made in the loopPop_UserService. Then we add that user information (firstname, loopPopUser) to the model, which can be used again for the frontend. We then "main", which means that we're giving the main.html page they wanted.

```java
private String getCurrentAuthenticatedUsername() {
    // Get the authentication object from the security context
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    if (authentication != null && authentication.isAuthenticated()
            && !(authentication instanceof AnonymousAuthenticationToken)) {
        // Return the authenticated user's username
        return authentication.getName();
    }
    return "Guest";
}
```

This part of the code to take the authenticated user's username and to be displayed in the main page. The "hello (Username)" greetings . But due to several reasons, I decided not to use this method.

```java
@PostMapping("/update-profile")
public String updateProfile(@ModelAttribute("loopPopUser")
                                LoopPop_User updatedLoopPopUser,
                            @AuthenticationPrincipal UserDetails userDetails) {
    if (userDetails == null) {
        // Redirect to the login page if the user is not authenticated
        return "redirect:/login";
    }

    String email = userDetails.getUsername();
    LoopPop_User existingUser = loopPop_UserService.findByEmail(email);

    if (existingUser != null) {
        // Update the user's profile with the provided data
        Long userId = existingUser.getId();
        loopPop_UserService.update_LoopPop_User(userId, updatedLoopPopUser);
    }

    // Redirect to the main page with a success query parameter
    return "redirect:/main?success";
}
```

This PostMapping request endpoint is for the path /update-profile and is used when a user wants to update their profile. Like the previous code, if the user isn't authenticated, the user will be redirected back to the login page.

It then stores the information of the user name and email into the variable, this is to check the profile existence. If the process resulted in a success, it will be redirected to :/main?success.

```java
@GetMapping("/profile/{userId}")
public String getProfilePage(@PathVariable Long userId, Model model) {
    // Find the user by their ID and add them to the model
    LoopPop_User user = loopPop_UserService.findById(userId);
    model.addAttribute( attributeName: "user", user);
    return "profile";
}
```

This code function is to enable the feature for users to view their profile and other people's profile. Basically the backbone for the user profile viewing feature in the comment . We find the userID by .findById to find the ID of the user in our database, and the modal is also for the frontend. We then return the profile page.

RegistrationRequest

```java
4 usages  Justin Hadinata
public class RegistrationRequest {
    3 usages
    private String firstname;
    3 usages
    private String lastname;
    3 usages
    private String email;
    3 usages
    private String password;

    1 usage  Justin Hadinata
    public String getFirstname() { return this.firstname; }

    1 usage  Justin Hadinata
    public String getLastname() { return this.lastname; }

    Justin Hadinata
    public String getEmail() { return this.email; }
```

```
1 usage   Justin Hadinata
public String getLastname() { return this.lastname; }
Justin Hadinata
public String getEmail() { return this.email; }
1 usage   Justin Hadinata
public String getPassword() { return this.password; }
no usages   Justin Hadinata
public void setFirstname(String firstname) { this.firstname = firstname; }
no usages   Justin Hadinata
public void setLastname(String lastname) { this.lastname = lastname; }
Justin Hadinata
public void setEmail(String email) { this.email = email; }
no usages   Justin Hadinata
public void setPassword(String password) { this.password = password; }
Justin Hadinata *
public String toString() {
    return "RegistrationRequest{firstname='" +
            this.firstname + "', lastname='" + this.lastname + "', " +
            "email='" + this.email + "', password='" + this.password + "'}";
```

The RegistrationRequest acts as the DTO (Data-Transfer-Object), whose purpose is to encapsulate data that is transferred between layers or components of an application. This type of class usually has the attributes, getter and setter, constructor, and the toString method.

```
@Service
public class RegistrationService {

    // Dependency that will be injected
    2 usages
    private final AppUserService appUserService;
```

Similar to @Controller, the @Service is used to indicate the class function in the Spring Boot architectures and layers. The @Service annotation is telling Spring boot that this class is part of the business service layer. In the code we also injected dependency from the AppUserService.

```java
// Constructor for dependency injection
no usages  Justin Hadinata
public RegistrationService(AppUserService appUserService) { this.appUserService = appUserService; }

// Method to register a new user
1 usage  Justin Hadinata *
public boolean register(RegistrationRequest request) {
    // Create a new AppUser object using the data from the registration request
    AppUser newUser = new AppUser(
            request.getFirstname(),
            request.getLastname(),
            request.getEmail(),
            request.getPassword(),
            AppUserRole.USER // Set the user role to USER
    );

    // Delegate the user sign-up process to the appUserService
    return appUserService.signUpUser(newUser);
}
```

We then make a constructor for the dependency injection like usual, and then we make a new method to register users by creating a new object. This method will be used in the Controller class, and we also role to the user. We then return the object.

AppUser:

AppUser

```java
@Entity
@Table(name = "app_user")
public class AppUser implements UserDetails {

1 usage
    @SequenceGenerator(
            name = "user_sequence",         // Name of the sequence generator
            sequenceName = "user_sequence", // Name of the database sequence
            allocationSize = 1              // Increment size for the sequence
    )
    // Specify the primary key of the entity
    @Id
    // Define the strategy for generating the primary key value
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE, // Use sequence-based ID generation
            generator = "user_sequence"         // Specify the sequence generator to use
    )
    private Long id;
```

The @Entity means that the class is mapped to a database, by marking it as a JPA entity (All of this will be used for our SQL database). The @Table is used to specify the table name in the database, which is app_user.

The Implement UserDetails is for the user authentication process.

The @SequenceGenerator is used for our SQL database, because the web-application utilizes relational databases. The @Sequence, @SequenceGenerator, @Id, and @Generated Value is basically to generate the unique identifier for the table, which can also be called as primary key if I remember correctly.

```
private String firstName;
2 usages
private String lastName;
5 usages
private String email;
3 usages
private String password;

// Specify that the appUserRole field should be persisted as a String in the database
5 usages
@Enumerated(EnumType.STRING)
private AppUserRole appUserRole;

// Define fields for account status with default values
4 usages
private Boolean locked = false;   // Indicates if the account is locked
4 usages
private Boolean enabled = true;   // Indicates if the account is enabled
```

The private attributes that will be used for the authentication is declared here, and the @Enumerated means that the appUserRole field should be persisted as a string in the database.

```
// Constructor to initialize all fields except id
1 usage  Justin Hadinata
public AppUser(String firstName, String lastName, String email, String password, AppUserRole appUserRole) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.password = password;
    this.appUserRole = appUserRole;
}

// Default constructor for JPA
no usages  Justin Hadinata
public AppUser() {
}
```

The constructor for the AppUser, you can see that in the constructor there is no ID, this is because the ID will be generated with @Sequence.

```java
// Return the authorities granted to the user (required by UserDetails)
Justin Hadinata *
public Collection<? extends GrantedAuthority> getAuthorities() {
    // Create a SimpleGrantedAuthority object with the name of the user's role
    SimpleGrantedAuthority authority = new SimpleGrantedAuthority(this.appUserRole.name());
    // Return the authority as a collection
    return Collections.singletonList(authority);
}
```

We then return the authorities granted to the user, which is for the UserDetails.

We can pass them as singletonList.

```java
Justin Hadinata
public Long getId() { return this.id; }

4 usages  Justin Hadinata
public String getFirstName() { return this.firstName; }

4 usages  Justin Hadinata
public String getLastName() { return this.lastName; }

Justin Hadinata
public String getPassword() { return this.password; }

// Return the username for authentication (in this case, the email)
Justin Hadinata
public String getUsername() { return this.email; }

// Account non-expired status (always true in this case)
Justin Hadinata
public boolean isAccountNonExpired() { return true; }

// Account non-locked status (true if the account is not locked)
9 usages  Justin Hadinata
public boolean isAccountNonLocked() { return !this.locked; }
```

```java
// Account enabled status
Justin Hadinata
public boolean isEnabled() { return this.enabled; }

// Setter for the password
1 usage  Justin Hadinata
public void setPassword(String password) { this.password = password; }
```

All the getter and setter, and declaring some public boolean attribute for the equals method.

```
public boolean equals(final Object o) {
    if (o == this) {
        return true;
    } else if (!(o instanceof AppUser)) {
        return false;
    } else {
        AppUser other = (AppUser)o;
        if (!other.canEqual( other: this)) {
            return false;
```

The equals method in the AppUser class in an overridden implementation of the Object.equals() method. The purpose of the method is to compare two AppUser objects based on their field values and then determine whether the object is equal or not.

The method takes an object as the parameter to compare it with any other object, it first checks for reference equality using ==, and then use the instanceof to ensure that object that is being compared is also an AppUser object. The canEqual method is called to allow for a equality test.

```
    return false;
} else {
    label107: {
        Object this$id = this.getId();
        Object other$id = other.getId();
        if (this$id == null) {
            if (other$id == null) {
                break label107;
            }
        } else if (this$id.equals(other$id)) {
            break label107;
        }

        return false;
    }
```

```
Object this$locked = this.locked;
Object other$locked = other.locked;
if (this$locked == null) {
    if (other$locked != null) {
        return false;
    }
} else if (!this$locked.equals(other$locked)) {
    return false;
}

Object this$enabled = this.enabled;
Object other$enabled = other.enabled;
if (this$enabled == null) {
    if (other$enabled != null) {
        return false;
    }
} else if (!this$enabled.equals(other$enabled)) {
    return false;
}
```

```
} else if (!this$enabled.equals(other$enabled)) {
    return false;
}

label86: {
    Object this$firstName = this.getFirstName();
    Object other$firstName = other.getFirstName();
    if (this$firstName == null) {
        if (other$firstName == null) {
            break label86;
        }
    } else if (this$firstName.equals(other$firstName)) {
        break label86;
    }

    return false;
}
```

```
label79: {
    Object this$lastName = this.getLastName();
    Object other$lastName = other.getLastName();
    if (this$lastName == null) {
        if (other$lastName == null) {
            break label79;
        }
    } else if (this$lastName.equals(other$lastName)) {
        break label79;
    }

    return false;
}

label72: {
    Object this$email = this.email;
    Object other$email = other.email;
    if (this$email == null) {
        if (other$email == null) {
            break label72;
        }
    } else if (this$email.equals(other$email)) {
        break label72;
```

```
        return false;
    }

    Object this$password = this.getPassword();
    Object other$password = other.getPassword();
    if (this$password == null) {
        if (other$password != null) {
            return false;
        }
    } else if (!this$password.equals(other$password)) {
        return false;
    }

    Object this$appUserRole = this.appUserRole;
    Object other$appUserRole = other.appUserRole;
    if (this$appUserRole == null) {
        if (other$appUserRole != null) {
            return false;
        }
    } else if (!this$appUserRole.equals(other$appUserRole)) {
        return false;
    }

    return true;
}
```

Each of the fields in each object is being compared, as you can see in the code. The object is being compared in the firstname, lastname, id, password, and more.

```
1 usage  Justin Hadinata
public void setPassword(String password) { this.password = password; }
```

This is the setter for the password attribute

```
public int hashCode() {
    boolean PRIME = true;
    int result = 1;
    Object $id = this.getId();
    result = result * 59 + ($id == null ? 43 : $id.hashCode());
    Object $locked = this.locked;
    result = result * 59 + ($locked == null ? 43 : $locked.hashCode());
    Object $enabled = this.enabled;
    result = result * 59 + ($enabled == null ? 43 : $enabled.hashCode());
    Object $firstName = this.getFirstName();
    result = result * 59 + ($firstName == null ? 43 : $firstName.hashCode());
    Object $lastName = this.getLastName();
    result = result * 59 + ($lastName == null ? 43 : $lastName.hashCode());
    Object $email = this.email;
    result = result * 59 + ($email == null ? 43 : $email.hashCode());
    Object $password = this.getPassword();
    result = result * 59 + ($password == null ? 43 : $password.hashCode());
    Object $appUserRole = this.appUserRole;
    result = result * 59 + ($appUserRole == null ? 43 : $appUserRole.hashCode());
    return result;
}
```

This hashing algorithm is used for hashing the password of our user. Note that hashing is different with encryption, as hashing is a one-way encryption, meaning that after a value or information is being hashed, the value cannot be read anymore, or unhashed.

AppUserRepository:

```
@Repository
// Indicate that the methods of this repository should be executed within a transaction context
// with read-only transactions by default
@Transactional(readOnly = true)
public interface AppUserRepository extends JpaRepository<AppUser, Long> {

    // Define a method to find an AppUser by their email address
    // Spring Data JPA will automatically implement this method based on the method name
    3 usages  Justin Hadinata
    Optional<AppUser> findByEmail(String email);
}
```

The @Repository is similar to the previous annotation, as it marks the class to a specific class layer. In this way, it indicates that the class is part of a JPA repository, which is the Data Access Layer. In my code, if the filename ends with a repository, it means that the class is a Data Access Layer. The @Transactional means the transactions are by default read-only.

This is an Interface, rather than a class. It defines a method to find an AppUser, and the Spring Data JPA will automatically utilize it.

AppUserRole

```
package com.LoopPop.LoopPop.AppUser;

6 usages  Justin Hadinata
public enum AppUserRole {
    1 usage
    USER,
    1 usage
    ADMIN

}
```

This class is used to define an enumeration named AppUserRole (Used in the AppUser DTO), and the USER represents a regular user role, while admin represents an administrative user role.

AppUserService

```java
// Mark this class as a Spring service component
6 usages  Justin Hadinata *
@Service
public class AppUserService implements UserDetailsService {

    // Constant message template for user not found exception
    1 usage
    private final static String USER_NOT_FOUND = "user with email %s not found";

    // Dependencies that will be injected
    5 usages
    private final AppUserRepository appUserRepository;
    2 usages
    private final BCryptPasswordEncoder bCryptPasswordEncoder;
    2 usages
    private final LoopPop_UserService loopPop_UserService;
```

@Service annotation is used again here because this class is also part of a Business Service layer. We declare a message for the user not found exception, and then we declare all of the dependencies that we wanted to inject.

```java
@Autowired
public AppUserService(AppUserRepository appUserRepository, LoopPop_UserService loopPop_UserService,
                      BCryptPasswordEncoder bCryptPasswordEncoder) {
    this.appUserRepository = appUserRepository;
    this.loopPop_UserService = loopPop_UserService;
    this.bCryptPasswordEncoder = bCryptPasswordEncoder;
}

// Override the method to load user details by username (email in this case)
Justin Hadinata *
@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    // Find the user by email or throw an exception if not found
    return appUserRepository.findByEmail(email)
            .orElseThrow(() -> new UsernameNotFoundException(String.format(USER_NOT_FOUND, email)));
}

// Method to find a user by email and return the AppUser object or null if not found
no usages  Justin Hadinata
public AppUser findUserByEmail(String email) { return appUserRepository.findByEmail(email).orElse( other: null); }
```

Then we make a constructor for those dependency injection, using the @Autowire so that Spring knows where to inject the dependency to.

The @Override annotation is to override the method to load user details by username, which is the email in this case. We return the email through the findByEmail, and we make a throw error method to catch an error.

We then make the findUserByEmail method which is utilizing the methods in the appUserRepository.

```
public boolean signUpUser(AppUser appUser) {
    // Check if the user already exists by email
    boolean userExists = appUserRepository.findByEmail(appUser.getUsername()).isPresent();

    if (userExists) {
        return userExists; // If user exists, return true
    }

    // Encode the user's password
    String encodedPassword = bCryptPasswordEncoder.encode(appUser.getPassword());
    appUser.setPassword(encodedPassword);

    // Save the new user in the repository
    appUserRepository.save(appUser);

    // Create a new LoopPop_User entity with the same email and name as the AppUser entity
    LoopPop_User loopPop_User = new LoopPop_User();
    loopPop_User.setEmail(appUser.getUsername());
    loopPop_User.setName(appUser.getFirstName() + " " + appUser.getLastName());
    loopPop_User.setDob(LocalDate.now()); // Set a default value for the dob field

    // Save the new LoopPop_User entity
    loopPop_UserService.addNew_LoopPop_User(loopPop_User);

    return userExists; // Return false indicating the user did not exist previously
```

This is the signUpUser method that is used in our registration request, first it checks where the user already exists through the findByEmail method. If it exists, it will return the userExists value, and if not, it will encode the user's password and then save the new user in the repository.

Then we create a newPop_User entity with the same email and name as the AppUser entity, and we save the. We then return userExists, indicating that the user now exists.

Comment:

Comment

```java
@Entity
public class Comment {

    // Define a logger for this class
    4 usages
    private static final Logger logger = LoggerFactory.getLogger(Comment.class);

    // Specify the primary key of the entity and define the strategy for generating the primary key value
    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Define fields to map to columns in the comment table
    3 usages
    private String content;
    3 usages
    private String tag;

    // Define a many-to-one relationship between Comment and LoopPop_User
    // Each comment is associated with one user, but a user can have many comments
```

```java
3 usages
@ManyToOne
@JoinColumn(name = "user_id", nullable = false) // Specify the foreign key column and make it non-nullable
private LoopPop_User user;

// Define a method to log the fields of the entity
// This method will be called after the entity is loaded, persisted, or updated
no usages  Justin Hadinata *
@PostLoad
@PostPersist
@PostUpdate
private void logFields() {
    logger.info("id: {}", id);         // Log the id field
    logger.info("content: {}", content); // Log the content field
    logger.info("tag: {}", tag);       // Log the tag field
    logger.info("user: {}", user);     // Log the user field
}

// Setters and getters for the fields
```

The comment DTO is also the same as the AppUser one, and the annotation logic is also the same. The only difference is one I have a Logger for debugging, and @ManyToOne, and @JoinColumn.

The @ManyToOne is used so that a user entity, can create many comments entity, And the @JoinColumn is used to specify the foreign key, so it can connect with other tables.

Foreign Key -> A primary key in another table, and is used to connect with

another table, hence the name relational database. The @PostLoad and @PostPersist and @PostUpdate are for debugging.

```
public Long getId() { return id; }
Justin Hadinata
public void setId(Long id) { this.id = id; }

Justin Hadinata
public String getContent() { return content; }

Justin Hadinata
public void setContent(String content) { this.content = content; }

no usages  Justin Hadinata
public String getTag() { return tag; }

1 usage  Justin Hadinata
public void setTag(String tag) { this.tag = tag; }

Justin Hadinata
public LoopPop_User getUser() { return user; }

Justin Hadinata
public void setUser(LoopPop_User user) { this.user = user; }
```

This part of the code serves as the getter and setter, the same as the previous other code.

CommentService

```
@Service
public class CommentService {

    2 usages
    @Autowired
    private CommentRepository commentRepository;

    1 usage
    @Autowired
    private LoopPop_UserRepository userRepository;

    1 usage  Justin Hadinata
    public Comment addComment(Comment comment, Long userId) {
        LoopPop_User user = userRepository.findById(userId)
                .orElseThrow(() -> new IllegalArgumentException("Invalid user ID"));
        comment.setUser(user);
        return commentRepository.save(comment);
    }

    1 usage  Justin Hadinata
    public List<Comment> getAllComments() { return commentRepository.findAll(); }
}
```

Same as before the @Service annotation is used to mark the class as a business service layer.

There are two methods, addComment and getAllComment. AddComment is a method to add comments, but it will add it for a specific user, this will be important to view the commentator's profile .

It first checks the user by theirID, and throws an exception if the user id is not found in the database. Using getter and setter, we then set the comment with the found user and save the comment in the repository and then return the saved comment.

The GetAllComments will retrieve all of the comments from the repository.

CommentController

```
@RestController
// Specify the base path for all endpoints in this controller
@RequestMapping(path = "api/v1/comments")
public class CommentController {

    // Dependencies that will be injected
    3 usages
    private final CommentService commentService;
    2 usages
    private final LoopPop_UserService userService;

    // Constructor for dependency injection
    no usages  Justin Hadinata
    @Autowired
    public CommentController(CommentService commentService, LoopPop_UserService userService) {
        this.commentService = commentService;
        this.userService = userService;
    }
```

The @RestController annotation will mark the class as a Rest Controller, which handles the HTTP Request.

The @RequestMapping(path =”api/v1/comments”) in the code, means the CRUD request, or get and post mapping request have to be traveled on that specific endpoint. People usually can conduct API testing with Postman, and set the path

in the Postman. Same as before, we declared dependencies that we want to inject, and then we create the constructor for every single one of theme.

```java
@PostMapping
public ResponseEntity<?> addComment(@RequestBody CommentRequest commentRequest, @AuthenticationPrincipal UserDetails userDetails) {
    if (userDetails == null) {
        // Return a JSON response for unauthorized access if the user is not authenticated
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body(Collections.singletonMap("error", "Unauthorized"));
    }
    // Retrieve the user based on their email (username)
    LoopPop_User user = userService.findByEmail(userDetails.getUsername());
    if (user == null) {
        // Return a JSON response for user not found if the user does not exist
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body(Collections.singletonMap("error", "User not found"));
    }
    // Create a new Comment object and set its fields based on the request
    Comment comment = new Comment();
    comment.setContent(commentRequest.getContent());
    comment.setTag(commentRequest.getTag());
    comment.setUser(user); // Associate the comment with the retrieved user

    // Save the new comment using the comment service and return it in the response
    Comment newComment = commentService.addComment(comment, user.getId());
    return ResponseEntity.ok(newComment);
}
```

The @PostMapping annotation in this @RestController class defines an endpoint for adding new comments. When a POST request is made to the base path "/api/v1/comments", this method is invoked. It takes a Comment Request object in the request body and a User Details object from the authentication context. The method first checks if the user is authenticated. If not, it returns an unauthorized status with an error message. If authenticated, it retrieves the user from the database using their email. If the user is found, a new Comment object is created with the content and tag from the request, and associated with the authenticated user. This new comment is then saved using the Comment Service and returned in the response. This implementation ensures that only authenticated users can post comments and that each comment is correctly associated with its author.

```
no usages   Justin Hadinata *
@GetMapping
public List<Comment> getAllComments() {
    // Use the comment service to get all comments and return them
    return commentService.getAllComments();
}
```

The @GetMapping annotation in the commentController definition endpoint too for retrieving all comments. This method is just called the getAllComments method from the commentService.

CommentRepository

```
public interface CommentRepository extends JpaRepository<Comment, Long> {

    // Method to find comments by their content
    no usages   Justin Hadinata
    List<Comment> findByContent(String content);

    // Method to find comments by their tag
    no usages   new *
    List<Comment> findByTag(String tag);

    // Other query methods can be added here as needed for specific query requirements
}
```

The findByContent is used to find a list of comments by their content, and with their tag

CommentRequest

```
public class CommentRequest {
    2 usages
    private String content;
    2 usages
    private String tag;

    Justin Hadinata
    public String getContent() { return content; }

    Justin Hadinata
    public void setContent(String content) { this.content = content; }

    1 usage   Justin Hadinata
    public String getTag() { return tag; }

    no usages   Justin Hadinata
    public void setTag(String tag) { this.tag = tag; }
}
```

This is the DTO class, which holds the getter and setter, and the attribute.

LoopPop_User:

LoopPop_User (I will explain them through the comments instead)

```
// Mark this class as a JPA entity (a class that is mapped to a
database table)
@Entity
// Specify the details of the database table that this entity will
map to
@Table(name = "looppop_user")
public class LoopPop_User {

   // Primary key field mapped to the 'id' column in the database
   @Id
   // Configure a sequence generator for generating unique IDs
   @SequenceGenerator(
           name = "mainUser_sequence",         // Name of the
sequence generator
           sequenceName = "mainUser_sequence", // Name of the
database sequence
```

```
            allocationSize = 1                  // Increment size
for the sequence
    )
    // Specify the strategy for generating the primary key value
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE, // Use
sequence-based ID generation
            generator =  "mainUser_sequence"    // Name of the
sequence generator to use
    )
    private Long id;

    // Fields mapped to columns in the 'looppop_user' table
    private String name;
    private String email;
    private String hobby;
    private String favoriteMusic;

    // Column mapped to 'dob' in the database, not nullable,
default value is current date
    @Column(nullable = false, columnDefinition = "DATE DEFAULT
CURRENT_DATE")
    private LocalDate dob;

    // One-to-many relationship with Comment entities, mapped by
the 'user' field in Comment class
    @OneToMany(mappedBy = "user")
    private List<Comment> comments;

    // Transient field not persisted in the database
    @Transient
    private Integer age;

    // Constructors

    // Constructor with all fields
    public LoopPop_User(Long id, String name, String email, String
hobby, String favoriteMusic, LocalDate dob) {
```

```
        this.id = id;
        this.name = name;
        this.email = email;
        this.hobby = hobby;
        this.favoriteMusic = favoriteMusic;
        this.dob = dob;
    }

    // Constructor without 'id' field (typically used when creating
new entities)
    public LoopPop_User(String name, String email, String hobby,
String favoriteMusic, LocalDate dob) {
        this.name = name;
        this.email = email;
        this.hobby = hobby;
        this.favoriteMusic = favoriteMusic;
        this.dob = dob;
    }

    // Default constructor (required by JPA)
    public LoopPop_User() {
    }

    // Getters and Setters for all fields (required by JPA)

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```java
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getHobby() {
        return hobby;
    }

    public void setHobby(String hobby) {
        this.hobby = hobby;
    }

    public String getFavoriteMusic() {
        return favoriteMusic;
    }

    public void setFavoriteMusic(String favoriteMusic) {
        this.favoriteMusic = favoriteMusic;
    }

    public LocalDate getDob() {
        return dob;
    }

    public void setDob(LocalDate dob) {
        this.dob = dob;
    }

    // Calculate and return age based on 'dob' field
    public Integer getAge() {
        return Period.between(dob, LocalDate.now()).getYears();
```

```
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    // Override toString method to provide a string representation
of the object
    @Override
    public String toString() {
        return "LoopPop_User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", hobby='" + hobby + '\'' +
                ", favoriteMusic='" + favoriteMusic + '\'' +
                ", dob=" + dob +
                ", age=" + age +
                '}';
    }
}
```

LoopPop_UserConfig:

```
// Mark this class as a configuration class for Spring
@Configuration
public class LoopPop_UserConfig {

    // Define a bean for initializing data using CommandLineRunner
    @Bean
    CommandLineRunner commandLineRunner(LoopPop_UserRepository
repository) {
        return args -> {
            // Create instances of LoopPop_User with sample data
            LoopPop_User brad = new LoopPop_User(
                    1L, // ID
                    "Justin", // Name
                    "Brad@gmail.com", // Email
                    "Games", // Hobby
```

```
                    "Cranberries", // Favorite music
                    LocalDate.of(2000, Month.DECEMBER, 5) // Date
of birth
            );

            LoopPop_User jacob = new LoopPop_User(
                    "Jacob", // Name
                    "Jacob@gmail.com", // Email
                    "Basketball", // Hobby
                    "Gorrilaz", // Favorite music
                    LocalDate.of(2003, Month.DECEMBER, 29) // Date
of birth
            );

            // Save the created instances to the repository
            repository.saveAll(List.of(brad, jacob));
        };
    }
}
```

LoopPop_UserController:

```
@RestController
// Base path for all endpoints in this controller
@RequestMapping(path = "api/v1/looppop_user")
public class LoopPop_UserController {

    private final LoopPop_UserService loopPop_UserService;

    // Constructor injection to initialize LoopPop_UserService
    @Autowired
    public LoopPop_UserController(LoopPop_UserService
loopPopUserService) {
        this.loopPop_UserService = loopPopUserService;
    }

    // GET endpoint to fetch all LoopPop_User entities
    @GetMapping
    public List<LoopPop_User> GetLoopPopUser() {
```

```
        return loopPop_UserService.GetLoopPopUser();
    }

    // POST endpoint to register a new LoopPop_User
    @PostMapping
    public void registerNew_LoopPop_User(@RequestBody LoopPop_User
loopPop_user) {
        loopPop_UserService.addNew_LoopPop_User(loopPop_user);
    }

    // POST endpoint to update profile of a LoopPop_User
    @PostMapping("/main/profile")
    public void updateProfile(@RequestBody LoopPop_User
loopPop_user) {
        loopPop_UserService.addNew_LoopPop_User(loopPop_user);
    }

    // DELETE endpoint to delete a LoopPop_User by ID
    @DeleteMapping(path = "{LoopPop_UserId}")
    public void delete_LoopPop_user(@PathVariable("LoopPop_UserId")
Long LoopPop_UserId) {
        loopPop_UserService.LoopPop_deleteUser(LoopPop_UserId);
    }

    // PUT endpoint to update details of a LoopPop_User
    @PutMapping(path = "{LoopPop_UserId}")
    public void update_LoopPop_User(@PathVariable("LoopPop_UserId")
Long LoopPop_UserId,
                                    @RequestBody LoopPop_User
updatedLoopPop_User) {
        loopPop_UserService.update_LoopPop_User(LoopPop_UserId,
updatedLoopPop_User);
    }

    // POST endpoint to update profile of a LoopPop_User with
authentication
    @PostMapping("/update-profile")
```

```
   public String updateProfile(@ModelAttribute("loopPopUser")
LoopPop_User updatedLoopPopUser,
                                @AuthenticationPrincipal UserDetails
userDetails) {
       if (userDetails == null) {
           // Handle unauthorized access
           return "redirect:/login";
       }

       String email = userDetails.getUsername();
       LoopPop_User existingUser =
loopPop_UserService.findByEmail(email);

       if (existingUser != null) {
           Long userId = existingUser.getId();
           loopPop_UserService.update_LoopPop_User(userId,
updatedLoopPopUser);
       }

       // Redirect to a success page or display a success message
       return "redirect:/main?success";
   }

   // GET endpoint to fetch main page with user details
   @GetMapping("/main")
   public String mainPage(Model model, @AuthenticationPrincipal
UserDetails userDetails) {
       if (userDetails == null) {
           // Handle unauthorized access
           return "redirect:/login";
       }

       String email = userDetails.getUsername();
       LoopPop_User existingUser =
loopPop_UserService.findByEmail(email);

       if (existingUser != null) {
           model.addAttribute("loopPopUser", existingUser);
```

```
        }

        // Return the main page template name
        return "main";
    }

}
```

All of the CRUD requests in this controller aren't used, the reason being I didn't have the time to fully utilize all of the requests because of all the debugging I need to do.

LoopPop_UserRepository:

```
// Indicates that this interface is a repository for LoopPop_User entities
@Repository
public interface LoopPop_UserRepository extends JpaRepository<LoopPop_User, Long> {

    // Custom query to find a LoopPop_User by email using JPQL
    @Query("SELECT l FROM LoopPop_User l WHERE l.email = ?1")
    Optional<LoopPop_User> findsLoopPopUserByEmail(String email);
}
```

LoopPop_UserService:

```
@Service
public class LoopPop_UserService {

    private final LoopPop_UserRepository loopPop_userRepository;

    // Constructor injection to initialize LoopPop_UserRepository
    @Autowired
    public LoopPop_UserService(LoopPop_UserRepository loopPop_userRepository) {
        this.loopPop_userRepository = loopPop_userRepository;
    }
```

```
    // Method to fetch all LoopPop_User entities
    public List<LoopPop_User> GetLoopPopUser() {
        return loopPop_userRepository.findAll();
    }

    // Method to add a new LoopPop_User
    public void addNew_LoopPop_User(LoopPop_User loopPop_user) {
        Optional<LoopPop_User> loopPopUserOptional =
loopPop_userRepository.findsLoopPopUserByEmail(loopPop_user.getEma
il());
        if (loopPopUserOptional.isPresent()) {
            throw new IllegalStateException("Email is not
available");
        }
        loopPop_userRepository.save(loopPop_user);
    }

    // Method to delete a LoopPop_User by ID
    public void LoopPop_deleteUser(Long LoopPop_UserId) {
        boolean exists =
loopPop_userRepository.existsById(LoopPop_UserId);
        if (!exists) {
            throw new IllegalStateException("User with id " +
LoopPop_UserId + " does not exist!");
        }
        loopPop_userRepository.deleteById(LoopPop_UserId);
    }

    // Method to update details of a LoopPop_User
    @Transactional
    public void update_LoopPop_User(Long LoopPop_UserId,
LoopPop_User updatedLoopPop_User) {
        LoopPop_User loopPop_user =
loopPop_userRepository.findById(LoopPop_UserId)
                .orElseThrow(() -> new IllegalStateException("User
with id " + LoopPop_UserId + " does not exist!"));
```

```
        if (updatedLoopPop_User.getName() != null &&
!Objects.equals(loopPop_user.getName(),
updatedLoopPop_User.getName())) {
            loopPop_user.setName(updatedLoopPop_User.getName());
        }

        if (updatedLoopPop_User.getEmail() != null &&
!Objects.equals(loopPop_user.getEmail(),
updatedLoopPop_User.getEmail())) {
            Optional<LoopPop_User> loopPopUserOptional =
loopPop_userRepository.findsLoopPopUserByEmail(updatedLoopPop_User
.getEmail());
            if (loopPopUserOptional.isPresent()) {
                throw new IllegalStateException("Email is not
available");
            }
            loopPop_user.setEmail(updatedLoopPop_User.getEmail());
        }

        // Update hobby and favoriteMusic
        if (updatedLoopPop_User.getHobby() != null) {
            loopPop_user.setHobby(updatedLoopPop_User.getHobby());
        }

        if (updatedLoopPop_User.getFavoriteMusic() != null) {

loopPop_user.setFavoriteMusic(updatedLoopPop_User.getFavoriteMusic
());
        }
    }

    // Method to find a LoopPop_User by email
    public LoopPop_User findByEmail(String email) {
        return
loopPop_userRepository.findsLoopPopUserByEmail(email)
                .orElseThrow(() -> new
UsernameNotFoundException("User not found with email: " + email));
    }
```

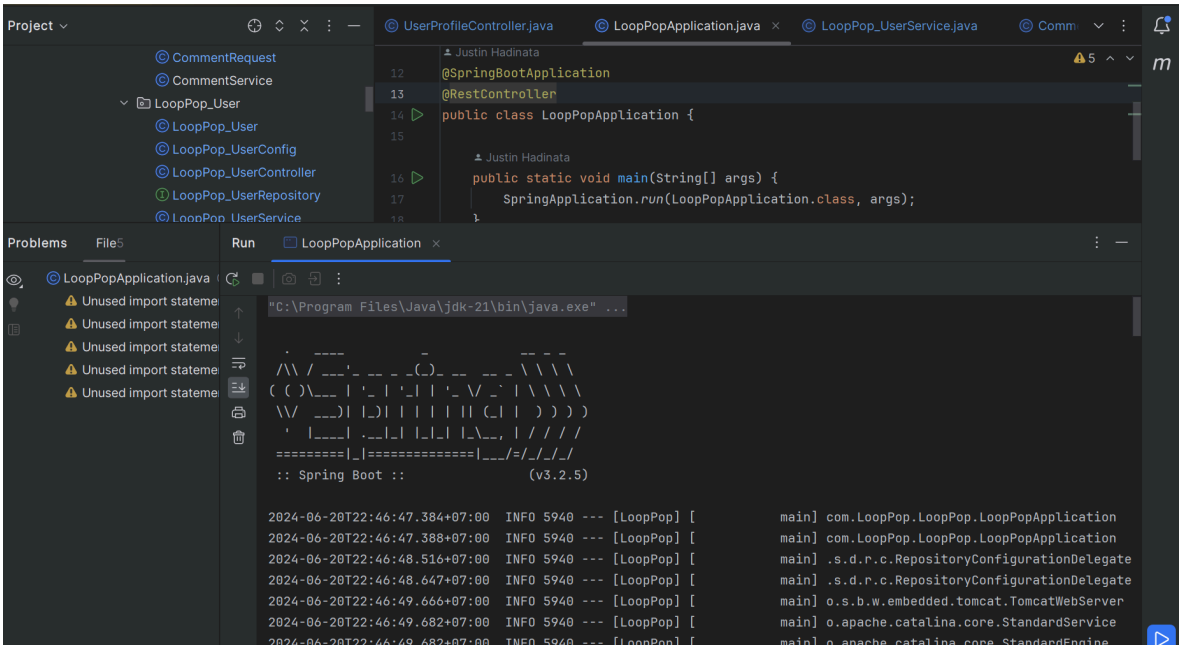
```
    // Method to find a LoopPop_User by ID
    public LoopPop_User findById(Long userId) {
        return loopPop_userRepository.findById(userId)
                .orElseThrow(() -> new IllegalStateException("User
with id " + userId + " does not exist!"));
    }
}
```

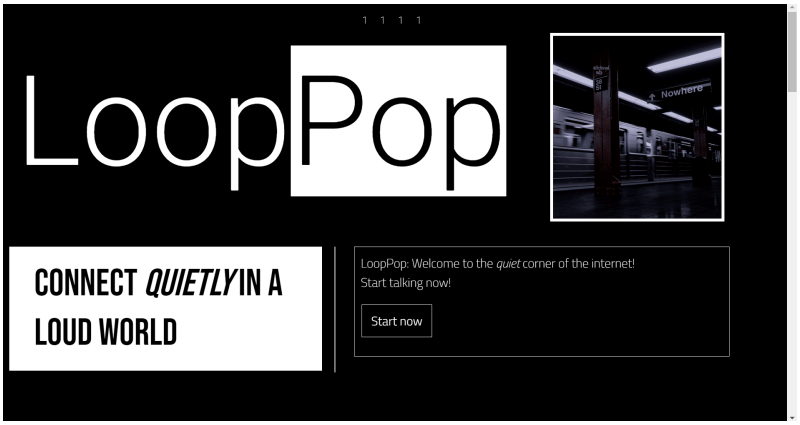
The User Controller -> for customization the User Profile

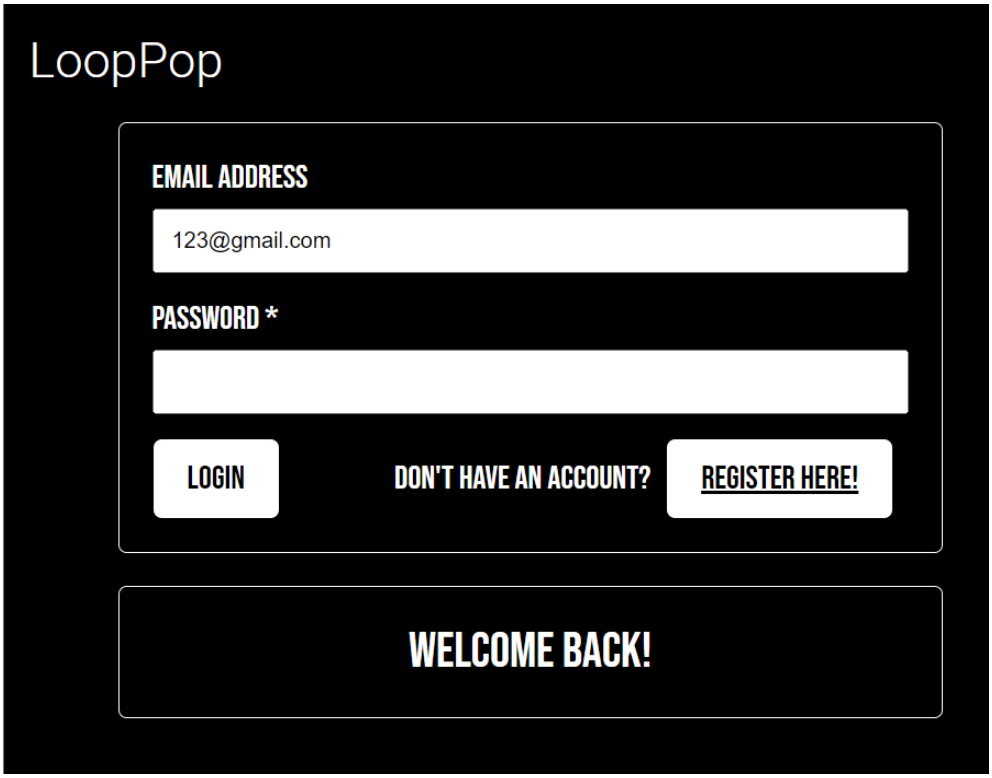
```
@Controller
@RequestMapping("/user")
public class UserProfileController {

    private final LoopPop_UserService userService;

    // Constructor injection to initialize LoopPop_UserService
    @Autowired
    public UserProfileController(LoopPop_UserService userService) {
        this.userService = userService;
    }

    // Handler method for displaying user profile
    @GetMapping("/profile")
    public String getProfile(Model model, @AuthenticationPrincipal
UserDetails userDetails) {
        if (userDetails == null) {
            return "redirect:/login"; // Redirect to login if user
is not authenticated
        }

        // Fetch user details based on authenticated username
        LoopPop_User user =
userService.findByEmail(userDetails.getUsername());
        model.addAttribute("user", user); // Add user object to the
model
```

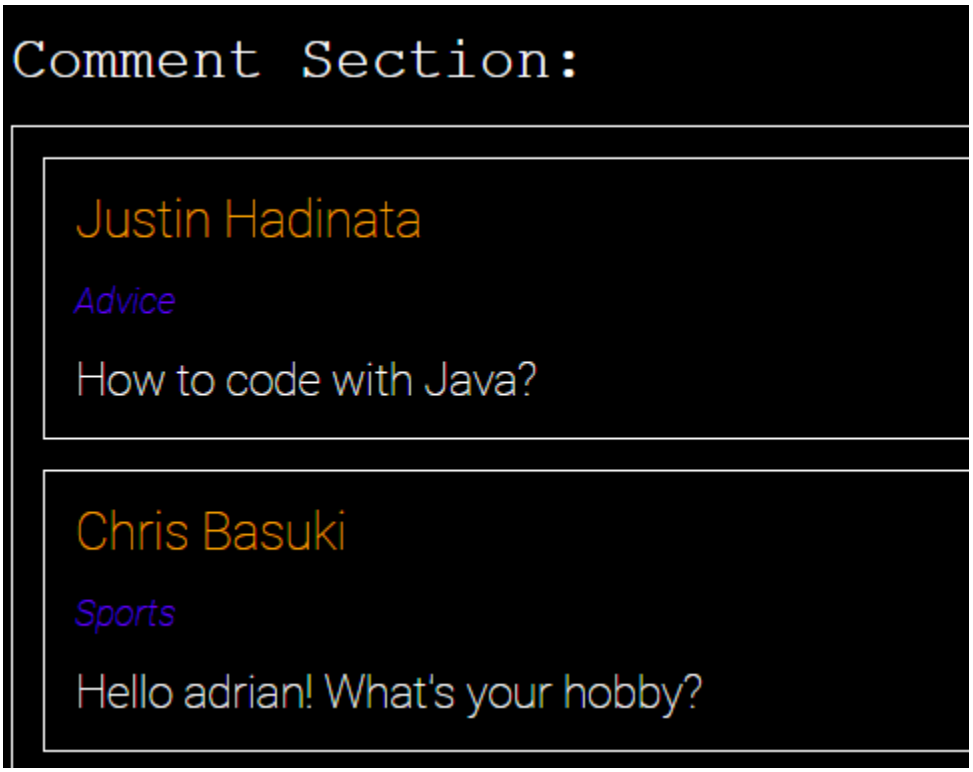
```
        return "profile"; // Return the profile template (make sure
this template exists)
    }

    // Handler method for updating user profile
    @PostMapping("/profile/update")
    public String updateProfile(@ModelAttribute LoopPop_User
updatedUser, @AuthenticationPrincipal UserDetails userDetails) {
        if (userDetails == null) {
            return "redirect:/login"; // Redirect to login if user
is not authenticated
        }

        // Fetch user details based on authenticated username
        LoopPop_User user =
userService.findByEmail(userDetails.getUsername());
        if (user != null) {
            // Update user's hobby and favorite music based on form
submission
            user.setHobby(updatedUser.getHobby());
            user.setFavoriteMusic(updatedUser.getFavoriteMusic());
            userService.update_LoopPop_User(user.getId(), user); //
Update user details in the database
        }

        return "redirect:/user/profile"; // Redirect to the profile
page after update
    }
}
```

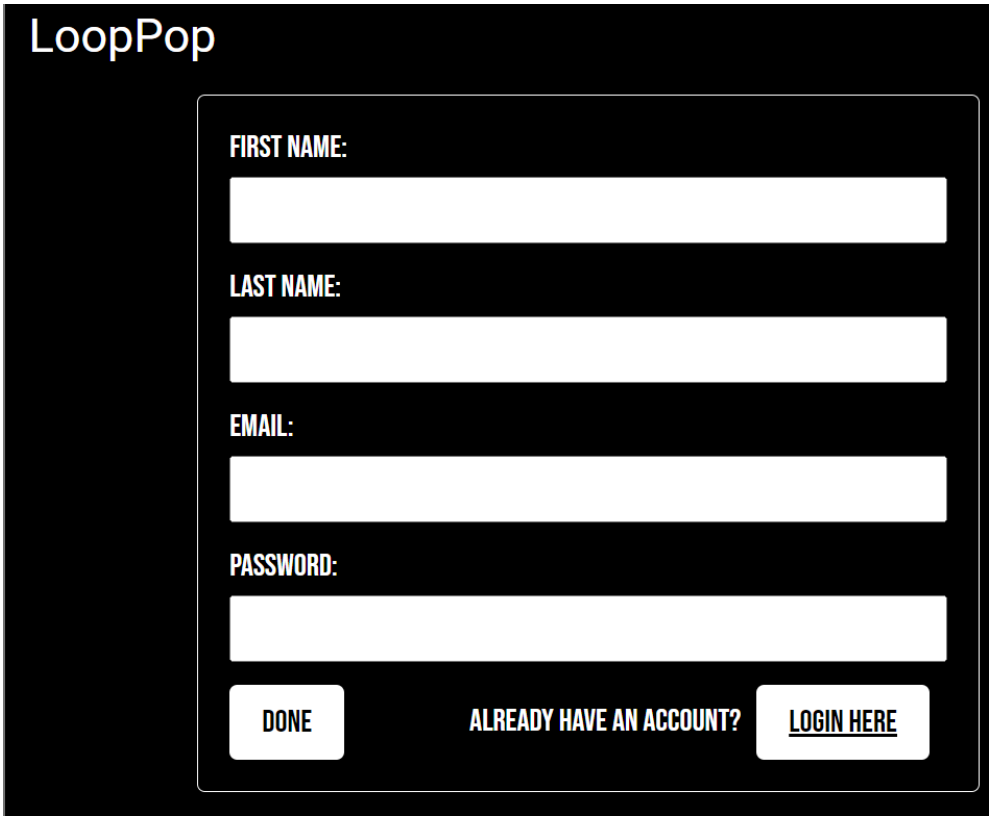
5. Screenshot of the program working

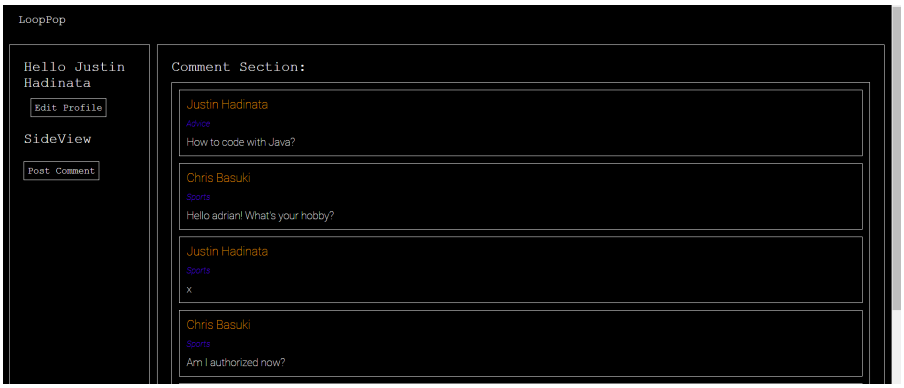


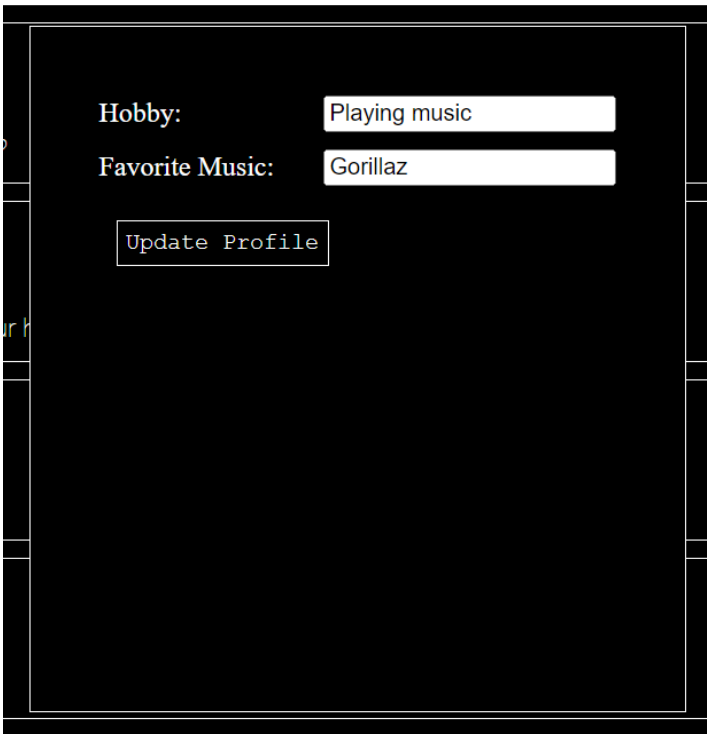



LoopPop
EMAIL ADDRESS
123@gmail.com
PASSWORD *
LOGIN
DON'T HAVE AN ACCOUNT?
REGISTER HERE!
WELCOME BACK!


Comment Section:
Justin Hadinata
Advice
How to code with Java?
Chris Basuki
Sports
Hello adrian! What's your hobby?


LoopPop
FIRST NAME:
LAST NAME:
EMAIL:
PASSWORD:
DONE
ALREADY HAVE AN ACCOUNT?
LOGIN HERE


LoopPop
Hello Justin Hadinata
Edit Profile
SideView
Post Comment
Comment Section:
Justin Hadinata
Advice
How to code with Java?
Chris Basuki
Sports
Hello adrian! What's your hobby?
Justin Hadinata
Sports
x
Chris Basuki
Sports
Am I authorized now?


Hobby:
Playing music
Favorite Music:
Gorillaz
Update Profile

6. The A1 size Poster