Session 1: In-Depth Guide for Authentication and Authorization

We'll dive deep into building a real-world user authentication system, guiding you step by step, with small subtasks and quizzes in between to ensure you fully understand each concept. By the end of this session, you'll be able to implement a basic sign-up and login flow with password security and token handling, all using Spring Boot. Let's make sure we know **why** we're doing each task!

---

1. Understanding Authentication and Authorization

Let's start by defining the two fundamental concepts of authentication and authorization.

- **Authentication**:
  `Have you ever needed an ID to enter a secure area?`
  - This is verifying ***who you are***. For example, when you log in to a website with an email and password, the system checks if you are who you claim to be.
  - On a website like Scaler, anyone can access public pages (e.g., event pages) without telling the site who they are.
  - However, to access secure content (e.g., user dashboards), users need to log in.
- **Authorization**:
  `Does having an ID allow you to access all areas of the building?`
  - This determines ***what you are allowed to do***. Even if you're authenticated, you may not have permission to access certain resources or perform certain actions.
  - Only hospital staff can enter an operating theater, even though many people can enter the hospital.
  - On a website, some pages (like admin dashboards) are restricted to authorized users only. This is managed via **roles**.

Why It Matters:

Understanding these concepts is crucial in designing secure applications. Mismanagement of authentication can lead to unauthorized access and data breaches.

TASK

Task 1: Implement basic authentication flow

Goal

Implement User Service which will have a basic sign-up and login flow without any password encryption yet. This will help us understand basic authentication flow.

Requirements

- Ensure we have a separate User Service which will handle sign-up, login and other authentication requirements.
- Implement following REST endpoints :
    - POST **/signup** : For user registration, take email and password as input, save the user in database
    - POST **/login** : For user authentication, check if email and password matches with a user in database

> Next section will contain the steps to complete this task. We suggest you to take some time and try to implement yourself first

---

Step 1: Create the Required Project Structure

We'll be following the **MVC architecture** to structure the project.

- **Create the following directories**:
    - `controllers/` → for controllers that handle incoming API requests.
    - `services/` → for service classes that implement business logic.
    - `models/` → for database entities.
    - `repositories/` → for repository interfaces to interact with the database.
    - `dtos/` → for dtos to handle request and response
    - `config/` → for security configurations (we will add this later when dealing with JWT and Spring Security).

Step 2: Create Required Models

1. User Model

The `User` entity will contain attributes like `id`, `email`, `password`, and `role`. This model will represent a user in the database.

```
@Entity
@JsonDeserialize(as = User.class)
public class User extends BaseModel {
    private String email;
    private String password;
}
```

2. BaseModel

Create a `BaseModel` for common fields like `createdAt` and `updatedAt`.

```
@MappedSuperclass
public class BaseModel {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Getters and Setters
}
```

Step 3: Create Required Controllers

AuthController

The `AuthController` will handle user signup and login.

You will need to create corresponding dtos for request and response. We have created dtos a lot of time previously.

```
@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private AuthService authService;

    @PostMapping("/sign_up")
    public ResponseEntity<SignUpResponseDto> signUp(@RequestBody SignUpRequestDt
        SignUpResponseDto response = new SignUpResponseDto();
        try {
            if (authService.signUp(request.getEmail(), request.getPassword())) {
                response.setRequestStatus(RequestStatus.SUCCESS);
            } else {
                response.setRequestStatus(RequestStatus.FAILURE);
            }
            return new ResponseEntity<>(response, HttpStatus.OK);
        } catch (Exception e) {
            response.setRequestStatus(RequestStatus.FAILURE);
            return new ResponseEntity<>(response, HttpStatus.CONFLICT);
        }
    }

    @PostMapping("/login")
    public ResponseEntity<LoginResponseDto> login(@RequestBody LoginRequestDto r
        LoginResponseDto response = new LoginResponseDto();
        try {
            authService.login(request.getEmail(), request.getPassword());

            response.setRequestStatus(RequestStatus.SUCCESS);
            return new ResponseEntity<>(
                    response , HttpStatus.OK
            );
        } catch (Exception e) {
            response.setRequestStatus(RequestStatus.FAILURE);
            return new ResponseEntity<>(
                    response , HttpStatus.BAD_REQUEST
            );
        }
    }
}
```

Step 4: Create Required Services

AuthService

The `AuthService` will contain the business logic for user signup and login.

```
@Service
public class AuthService {
    private UserRepository userRepository;

    public AuthService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public boolean signUp(String email, String password) throws UserAlreadyExist
        if (userRepository.findByEmail(email).isPresent()) {
            throw new UserAlreadyExistsException("User with email: " + email + "
        }
        User user = new User();
        user.setEmail(email);
        user.setPassword(password);
        userRepository.save(user);
        return true;
    }

    public boolean login(String email, String password) throws UserNotFoundExcep
        Optional<User> userOptional = userRepository.findByEmail(email);
        if (userOptional.isEmpty()) {
            throw new UserNotFoundException("User with email: " + email + " not
        }
        boolean matches = password.equals(userOptional.get().getPassword());
        if (matches) {
            return true;
        } else {
            throw new WrongPasswordException("Wrong password.");
        }
        return false;
    }

}
```

Step 5: Use UserRepository with JPA

Create a **UserRepository** interface to handle database interactions using JPA.

```
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);
    User save(User user);
}
```

The `findByEmail` method will be used to fetch the user during login, ensuring the correct user is retrieved for authentication.

Step 6: Test the Flow

1. **Setup Database**:
    - Ensure your database is connected and running (MySQL/PostgreSQL).
    - Add the necessary credentials in the `application.properties` file for Spring Boot.

2. **Test Signup**:
    - Use **Postman** or any API client to hit the `/auth/signup` endpoint with a valid `User` JSON object:

    ```
    {
      "email": "testuser@example.com",
      "password": "password123"
    }
    ```

    - Check if the user is successfully stored in the database.

3. **Test Login**:
    - Hit the `/auth/login` endpoint with the same user credentials.
    - Ensure the login flow returns a success response if credentials are valid and an unauthorized response if they are not.

MCQ Quiz:

1. What is the primary purpose of authentication in an application?
    - A) To log user activity.

    - B) To identify who the user is.
    - C) To determine what a user can access.
    - **Answer:** B

2. Authorization checks:

    - A) Who the user is.
    - B) Whether the user has permission to access a resource.
    - C) Whether the user has logged out.
    - **Answer:** B

---

2. Understanding Role-Based Access Control (RBAC)

What is RBAC?
RBAC stands for **Role-Based Access Control**. It's a method of controlling access to resources based on the roles assigned to a user. Instead of giving permissions to individual users, permissions are grouped under roles, and users are assigned these roles.

Think of it like a Company:
In a company, employees are assigned roles such as Manager, Developer, or HR. Each role comes with specific access permissions:

- A Manager can approve leaves.
- An HR representative can access employee details.
- A Developer can access the codebase.

Employees (users) don’t get individual permissions; they inherit the permissions based on their roles. This abstraction simplifies permission management. Instead of handling permissions for every employee, you manage them based on roles.

RBAC in the Context of a Web Application

Imagine you are building an **e-commerce platform** with multiple services and user roles. Some roles could be:

- **Admin**: Can manage the entire system (create/delete products, manage users).
- **Seller**: Can add, update, and delete products they own.
- **Customer**: Can browse products, add items to their cart, and make purchases.

In this model:

- A customer can't access admin pages or modify product data.

- A seller can't access the customer’s order history but can manage their own inventory.

Roles have predefined permissions to specific actions, making the application secure and manageable.

How Does RBAC Work in an Application?

1. **Define Roles**:
   Roles are predefined with a set of permissions. For example:
   - **Admin**: Can create, read, update, delete products and users.
   - **Seller**: Can create, update, delete their own products.
   - **Customer**: Can browse, buy, and view their own purchase history.

2. **Assign Users to Roles**:
   Each user is assigned one or more roles. For instance:
   - Alice is an **Admin**.
   - Bob is a **Seller**.
   - Charlie is a **Customer**.

3. **Enforce Permissions**:
   When a user tries to perform an action (e.g., deleting a product), the system checks the user's role. If the role has permission, the action is allowed. Otherwise, access is denied.

Examples of RBAC in Action

Example 1: Scaler Academy

- **Mentees** (students) can access learning material and attend classes.
- **Mentors** can view their mentee’s progress and give feedback.
- **Admins** can manage all users and content.
- A mentee cannot access the mentor’s dashboard, and a mentor cannot perform administrative tasks unless explicitly given the admin role.

Example 2: Hospital System

- **Doctor**: Can access patient medical records.
- **Nurse**: Can update basic patient information but cannot access sensitive medical history.

- **Receptionist**: Can only access appointment details and contact information.
- **Admin**: Can manage all users and patient records.

In this example, a doctor might have access to patient records that a nurse cannot modify. Similarly, a receptionist can't access sensitive health data. The system enforces these restrictions based on the role assigned to each user.

Why is RBAC Useful?

- **Scalability**: Easily manage permissions for hundreds of users by assigning roles instead of handling individual user permissions.
- **Security**: Restrict access to sensitive operations based on roles, preventing unauthorized actions.
- **Simplicity**: Administrators manage roles instead of configuring permissions for every user, saving time and reducing errors.

Task 2: Implement RBAC

1. **Create Role Model** :

    - We will create a `Role` entity for Role-Based Access Control (RBAC). Each user will have a role (like `USER`, `ADMIN`).

    ```
    @Entity
    @JsonDeserialize(as = Role.class)
    public class Role {

        private String name;

        // Getters and Setters
    }
    ```

2. **Add Roles to User Entity**:

    - Add a `role` field to the `User` class. User can have multiple different roles, therefore let's take a Set for storing roles.

    ```
    @ManyToMany(fetch = FetchType.EAGER)
    @JsonIgnore
    private Set<Role> roles = new HashSet<>();
    ```

We will use these roles later in the upcoming sessions. So wait for practical usage of Roles.

MCQ Quiz (In-between):

1. What does RBAC stand for?

   - A) Role-Based Activity Control.
   - B) Role-Based Access Control.
   - C) Resource-Based Access Control.
   - **Answer:** B

2. Which of the following would be an example of RBAC?

   - A) Logging in with a password.
   - B) Only admins can access `/admin/dashboard`.
   - C) Saving user data to a database.
   - **Answer:** B

We have a Problem!

The Problem: Stateless Applications and HTTP

Imagine you're running a large amusement park. Your park is so popular that you've had to implement some special rules to keep things running smoothly:

1. The park has no central computer system to track visitors (like stateless servers).
2. Each ride operator can't remember who has paid for entry (like HTTP being stateless).
3. For security reasons, ride operators aren't allowed to handle money or check IDs (like not wanting to send passwords with every request).

Now, how do you ensure that only paying customers can enjoy the rides without slowing everything down?

The Solution: Token-Based Authentication

The Amusement Park Wristband Analogy

Let's solve this problem the way a real amusement park would:

1. **Entry (Login):**

   - When visitors arrive, they go to the ticket booth (like logging in to a website).
   - They show their ID and pay for entry (providing username and password).

- The ticket seller verifies their identity and payment (server checks credentials).

2. **The Wristband (Token):**

   - After verification, visitors get a special wristband (the server generates a token).
   - This wristband has:
     - A unique design for the day (unique identifier)
     - The current date (issued at time)
     - An expiry time (when the park closes)
     - Maybe even the visitor's age for certain rides (additional user info)

3. **Using the Wristband (Token Usage):**

   - Visitors show their wristband at each ride (token sent with each request).
   - Ride operators quickly check if the wristband is valid and not expired (server validates token).
   - If valid, visitors can ride without further questions (server processes the request).

4. **Benefits:**

   - Fast entry to rides (improved performance)
   - No need to carry ID or money around the park (enhanced security)
   - Ride operators don't need to remember faces or names (stateless application)

5. **Security Measures:**

   - Wristbands are designed to be tamper-evident (tokens are digitally signed).
   - They expire at the end of the day (token expiration).
   - Special wristbands for multi-day passes (refresh tokens).

How This Applies to Web Applications

Now, let's translate this to the digital world:

1. **Login:**

   - User provides credentials to the server.
   - Server verifies and generates a unique token (like issuing a wristband).

2. **Token Contents:**

   - User ID (like the unique design)

   - Issue time and expiration (like the date and park closing time)
   - User roles or permissions (like age restrictions for rides)

3. **Using the Token:**

   - The token is sent with each request to the server.
   - Server quickly validates the token without needing to check the full user database.

4. **Benefits:**

   - Reduced server load (like faster lines at rides)
   - Improved security (no need to send passwords with every request)
   - Works well with stateless applications (like ride operators not needing to remember faces)

By implementing token-based authentication, you create a system that's efficient, secure, and scalable - just like a well-run amusement park!

Adding Token-Based Authentication to the Existing Flow

To enhance the login flow and introduce token-based authentication, we will generate a simple token that combines the user's email and password. Although this is not secure for production, it helps you understand how tokens work. We will break this task into smaller steps so learners can follow each stage easily.

Step-by-Step Task: Adding a Basic Token

Step 1: Update the `AuthService` to Generate a Token

In the login process, we will generate a token upon successful authentication. The token will be a simple combination of the email and password concatenated with a colon `:`. Later, we'll use more secure methods, but for now, let's stick with this basic implementation.

Here's how you can modify the `login()` method:

```
public String login(String email, String password) throws UserNotFoundException,
    Optional<User> userOptional = userRepository.findByEmail(email);
    if (userOptional.isEmpty()) {
        throw new UserNotFoundException("User with email: " + email + " not founc
    }

    User user = userOptional.get();
    if (password.equals(user.getPassword())) {
        // Generating a basic token using email and password concatenated
        String token = email + ":" + password;
        return token;
    } else {
        throw new WrongPasswordException("Wrong password.");
    }
}
```

- **Explanation**:
    - When the user's email and password match, we create a token by concatenating the email and password. This is returned as a token from the service.
    - This token can then be used in subsequent requests for authenticating the user.

Step 2: Modify the `AuthController` to Return the Token

Update the login API in the `AuthController` to return the generated token. Modify the `LoginResponseDto` to include a field for the token, and ensure the token is sent back to the client when the login is successful.

Here's the updated controller:

```
@PostMapping("/login")
public ResponseEntity<LoginResponseDto> login(@RequestBody LoginRequestDto reque
    LoginResponseDto response = new LoginResponseDto();
    try {
        String token = authService.login(request.getEmail(), request.getPassword

        // Setting the token in the response
        response.setToken(token);
        response.setRequestStatus(RequestStatus.SUCCESS);

        return new ResponseEntity<>(response, HttpStatus.OK);
    } catch (Exception e) {
        response.setRequestStatus(RequestStatus.FAILURE);
        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }
}
```

- **Explanation**:
    - After the `authService.login()` method returns a token, we include it in the response (`LoginResponseDto`) and send it back with the status `SUCCESS` .

Step 3: Update the `LoginResponseDto`

Since we are now returning a token, update the `LoginResponseDto` to include a field for the token.

```java
public class LoginResponseDto {
    private String token;
    private RequestStatus requestStatus;

    // Getters and setters
    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }

    public RequestStatus getRequestStatus() {
        return requestStatus;
    }

    public void setRequestStatus(RequestStatus requestStatus) {
        this.requestStatus = requestStatus;
    }
}
```

Step 4: Testing the Token-Based Flow

Now that we have implemented token generation, follow these steps to test the flow:

1. **Create a New User**:
    - Use the `/sign_up` API to create a user with an email and password.
    - Ensure that the user is saved to the database.
2. **Login as the User**:
    - Use the `/login` API to log in with the registered email and password.
    - Upon successful login, the server should return a token in the response.
3. **Verify the Token**:
    - Ensure that the token returned by the server is the combination of the email and password as expected.

Example Workflow

1. **Sign Up**:
    - Make a POST request to `/auth/sign_up` with:

```
{
  "email": "john.doe@example.com",
  "password": "password123"
}
```

- Response:

```
{
  "requestStatus": "SUCCESS"
}
```

2. **Login**:

    - Make a POST request to `/auth/login` with:

```
{
  "email": "john.doe@example.com",
  "password": "password123"
}
```

- Response:

```
{
  "token": "john.doe@example.com:password123",
  "requestStatus": "SUCCESS"
}
```

Step 5: Securing the Token (Future Steps)

As discussed, combining email and password to form a token is not secure. In future tasks, we will:

- Introduce more secure token mechanisms like **JWT** (JSON Web Tokens).
- Implement token validation to ensure that tokens are properly authenticated for each request.

Quiz: Check Your Understanding

1. **Why should passwords not be stored in plain text?**

    - A) To reduce storage space.
    - B) To prevent users from logging in easily.
    - C) To avoid data breaches and enhance security.
    - D) To increase application performance.

2. **What is the purpose of a token in authentication?**
    - A) To represent user sessions in a compact form.
    - B) To store user preferences.
    - C) To hash passwords.
    - D) To improve database read performance.

3. **Why is concatenating email and password for tokens considered insecure?**
    - A) It uses too much memory.
    - B) It reveals sensitive information in the token.
    - C) It makes the token too long.
    - D) It cannot be used with modern browsers.

Answers:

1. C
2. A
3. B

We have a Problem!

The Problem : Storing Passwords Directly is Risky

Let’s first discuss why storing passwords directly in the database can be dangerous:

Scenario 1: Database Hack

- Imagine if someone hacked into your database and could see users' **email and password** in plain text.
- This allows the hacker to log into accounts as any user. Worse, many users reuse the same password across multiple websites, so the hacker could use this information to access **other platforms** as well.

One Solution: Hashing the Password

- Instead of storing passwords as plain text, we can store a **hashed** version.
- Hashing transforms the password into a random-looking string. When a user logs in, we hash the input and compare it to the stored hash.
- **Problem**: If two users have the same password, the hashed values will also be the same, allowing the hacker to identify accounts with the same password.

Real Solution: Hashing + Salting

- This problem is solved by **salting**.
- Salting is adding unique data (like a user's email or a random string) to the password before hashing it, ensuring that even identical passwords generate different hashes.

Bcrypt Password Encoder

To protect passwords, we introduce **Bcrypt**, which is a hashing function that automatically handles salting and ensures even the same password creates a unique hash every time.

How Bcrypt Works:

1. **Sign-Up Process**:
    - When a user signs up, we use `bcrypt.encode(password)`.
    - This generates a hashed and salted version of the password, which we then store in the database.
    - The important part is: **each time Bcrypt encodes the same password, it will generate a different hash**.
    - This way, even if two users have the same password, their hashes will be different.

2. **Login Process**:
    - On login, we cannot simply re-hash the password and compare it to the stored hash since Bcrypt generates a new hash each time.
    - Instead, we use Bcrypt's **`verify()`** method.
    - `verify()` checks if a given password could produce the stored hash, returning `true` if it matches and `false` if it doesn't.

Why Bcrypt is Secure:

- **Salting**: It automatically adds a random value (salt) before hashing the password, making it impossible to precompute hashes for common passwords.
- **Slow Computation**: Bcrypt is intentionally slow, which makes it difficult for hackers to brute-force attack passwords.

How to Add Bcrypt to Your Code

Now that we understand why Bcrypt is crucial for password security, let's go step-by-step on how to integrate it into the existing `signUp` and `login` processes.

Task 1: Modify AuthService to Use Bcrypt

Step 1: Add Dependency

First, ensure your project has the necessary Bcrypt dependency in `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This dependency provides the `BCryptPasswordEncoder`.

Step 2: Update AuthService to Encode Password on Sign-Up

- Instead of saving the plain password, we will encode it using Bcrypt before storing it in the database.

Example:

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Service
public class AuthService {

    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEnco

    public boolean signUp(String email, String password) throws UserAlreadyExist
        if (userRepository.findByEmail(email).isPresent()) {
            throw new UserAlreadyExistsException("User with email: " + email + "
        }
        String hashedPassword = passwordEncoder.encode(password);  // Encrypt the
        User user = new User();
        user.setEmail(email);
        user.setPassword(hashedPassword);  // Save the hashed password
        userRepository.save(user);
        return true;
    }
}
```

Step 3: Update AuthService for Login

- During login, use the `passwordEncoder.matches()` method to verify the user's input password against the stored hash.

Example:

```
public boolean login(String email, String password) throws UserNotFoundException
    Optional<User> userOptional = userRepository.findByEmail(email);
    if (userOptional.isEmpty()) {
        throw new UserNotFoundException("User with email: " + email + " not founc
    }

    User user = userOptional.get();
    if (passwordEncoder.matches(password, user.getPassword())) {   // Use Bcrypt’:
        return true;
    } else {
        throw new WrongPasswordException("Wrong password.");
    }
}
```

Step 4: Test the Flow

- After implementing the changes, use Postman or curl to sign up with a password, then log in with the same password to see how the flow works with Bcrypt encryption.

Recap: Why Bcrypt?

- **Security**: Plain text passwords are a significant security risk. If a database is hacked, plain passwords can lead to breaches across multiple platforms.
- **Salting and Hashing**: Bcrypt combines hashing with salting to make it difficult for attackers to reverse-engineer passwords.
- **Bcrypt Features**: It produces a unique hash each time and provides a method to verify if a password matches a stored hash without revealing the original password.

Try to debug if something is not working out! We will discuss all doubts in the next lab session

Summary of Session 1

By now, you should have:

- Gained an in-depth understanding of authentication, authorization, and RBAC.
- Implemented a basic sign-up and login flow with password encryption using **Bcrypt**.
- Integrated **token-based authentication** for secure API access.
- Tested everything hands-on through API requests.

Final Task of Session 1: Extend the System

1. Extend your user model to include multiple roles (like `ADMIN`, `USER`, etc.).
2. Add more protected routes in your application based on the role. For example:
    - `/admin/dashboard`: Only accessible by users with the `ADMIN` role.
    - `/user/profile`: Accessible by authenticated users.

Session 1 Recap Quiz:

1. **What is the difference between Authentication and Authorization?**

    - A) Authentication checks who you are, while Authorization checks what you can access.
    - B) Authorization checks who you are, while Authentication checks what you can access.
    - **Answer**: A

2. **Why is Bcrypt used in password handling?**

    - A) It encrypts passwords.
    - B) It hashes passwords securely, making them harder to reverse.
    - **Answer**: B

3. **What is the benefit of Token-based authentication over session-based authentication?**

    - A) Tokens allow for stateless authentication, reducing server-side overhead.
    - B) Tokens are faster to generate than sessions.
    - **Answer**: A

4. **Which HTTP header is typically used to send tokens for authorization?**

    - A) `Content-Type`.
    - B) `Authorization`.
    - C) `Token`.
    - **Answer**: B

Next Steps

For the next session, we'll dive into **JWT (JSON Web Tokens)** and explore more advanced concepts in token-based authentication. Make sure your current implementation is working smoothly and ask any questions you may have in the lab session!