Building a Secure Web Application with Spring Boot and Spring Security

In today's digital age, web applications have become an integral part of our lives. From online banking to social networking, we rely on these platforms for various purposes. However, with the increasing dependency on web applications, the need for robust security measures has never been more critical.

This tutorial is designed to equip you with the knowledge and skills needed to create a web application that can protect sensitive data, authenticate users, and authorize access based on roles.

Why Security Matters

In the realm of web development, security is not merely an optional feature; it's an absolute necessity. Cyber threats, data breaches, and unauthorized access are prevalent concerns that can have severe consequences, both for businesses and individuals. That's why understanding and implementing security best practices are fundamental for any developer.

Our project focuses on implementing security mechanisms using two powerful Java frameworks: Spring Boot and Spring Security. Spring Boot simplifies the development of production-ready applications, while Spring Security provides a comprehensive framework for handling authentication, authorization, and protection against common security vulnerabilities.

What You Will Learn

This tutorial will guide you through the process of creating a secure web application step by step. By the end of this tutorial, you will have built a fully functional web application with the following key features:

User Authentication: Users will be able to register, log in, and maintain secure sessions within the application.

Role-Based Access Control: The application will distinguish between different user roles, such as regular users, moderators, and administrators, granting access based on these roles.

Password Encryption: User passwords will be securely hashed and stored to protect user data.

JWT (JSON Web Tokens): We'll use JWTs to facilitate secure user authentication and authorization for protected endpoints.

Data Validation: Input data will be validated to prevent common security vulnerabilities, such as SQL injection and cross-site scripting (XSS) attacks.

Prerequisites

Before diving into this tutorial, you should have a basic understanding of the following:

Java: You should be familiar with Java programming concepts.

Spring Boot: While not mandatory, prior knowledge of Spring Boot will be helpful.

Database Basics: A basic understanding of databases, particularly relational databases like MySQL.

Web Development: Familiarity with web development concepts (HTTP, RESTful APIs) is beneficial.

Let's Get Started

Now that you have a glimpse of what we aim to achieve, let's get started with the project setup. We'll guide you through every step, from setting up the development environment to deploying your secure web application.

Overview of Spring Boot JWT Authentication Demo App

We will build a Spring Boot application in that:

- User can signup new account, or login with username & password.
- By User’s role (admin, moderator, user), we authorize the User to access resource

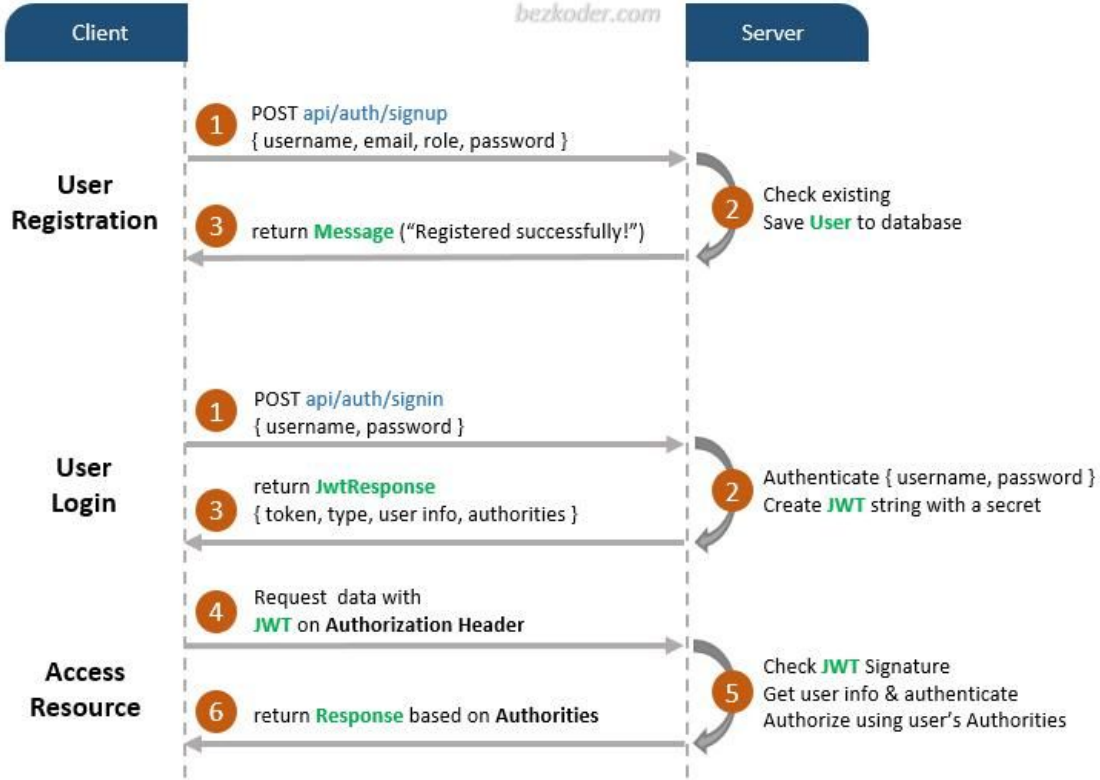
These are APIs that we need to provide

Methods	Urls	Actions
POST	/api/auth/signup	signup new account
POST	/api/auth/signin	login an account
GET	/api/test/all	retrieve public content
GET	/api/test/user	access User's content
GET	/api/test/mod	access Moderator's content
GET	/api/test/admin	access Admin's content

Spring Boot Signup & Login with JWT Authentication Flow

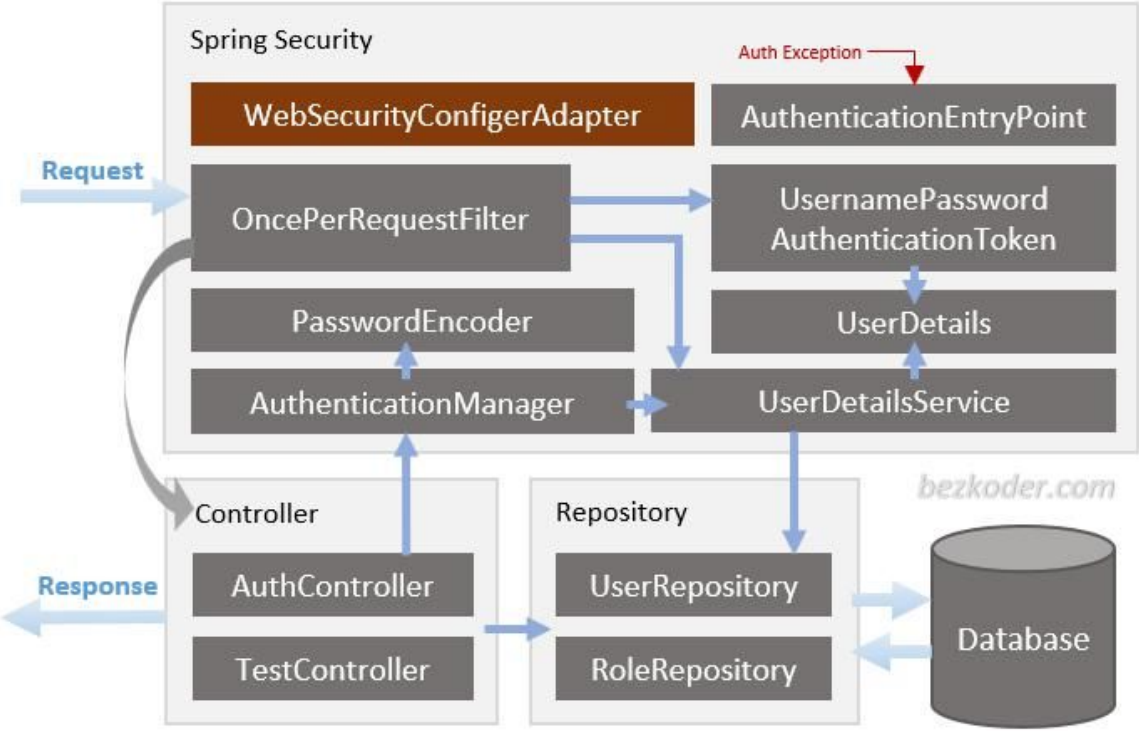
The diagram shows flow of how we implement User Registration, User Login and Authorization process.

A legal JWT must be added to HTTP Authorization Header if Client accesses protected resources.



Spring Boot Server Architecture with Spring Security

You can have an overview of our Spring Boot Server with the diagram below:



Now I will explain it briefly.

Spring Security

– WebSecurityConfigurerAdapter is the crux of our security implementation. It provides HttpSecurity configurations to configure cors, csrf, session management, rules for protected resources. We can also extend and customize the default configuration that contains the elements below.

– UserDetailsService interface has a method to load User by *username* and returns a `UserDetails` object that Spring Security can use for authentication and validation.

– `UserDetails` contains necessary information (such as: username, password, authorities) to build an Authentication object.

– UsernamePasswordAuthenticationToken gets {username, password} from login Request, `AuthenticationManager` will use it to authenticate a login account.

– AuthenticationManager has a `DaoAuthenticationProvider` (with help of `UserDetailsService` & `PasswordEncoder`) to validate `UsernamePasswordAuthenticationToken` object. If successful, `AuthenticationManager` returns a fully populated Authentication object (including granted authorities).

– OncePerRequestFilter makes a single execution for each request to our API. It provides a `doFilterInternal()` method that we will implement parsing & validating JWT, loading User details (using `UserDetailsService`), checking Authorizaion (using `UsernamePasswordAuthenticationToken`).

– AuthenticationEntryPoint will catch authentication error.

Repository contains `UserRepository` & `RoleRepository` to work with Database, will be imported into **Controller**.

Controller receives and handles request after it was filtered by `OncePerRequestFilter`.

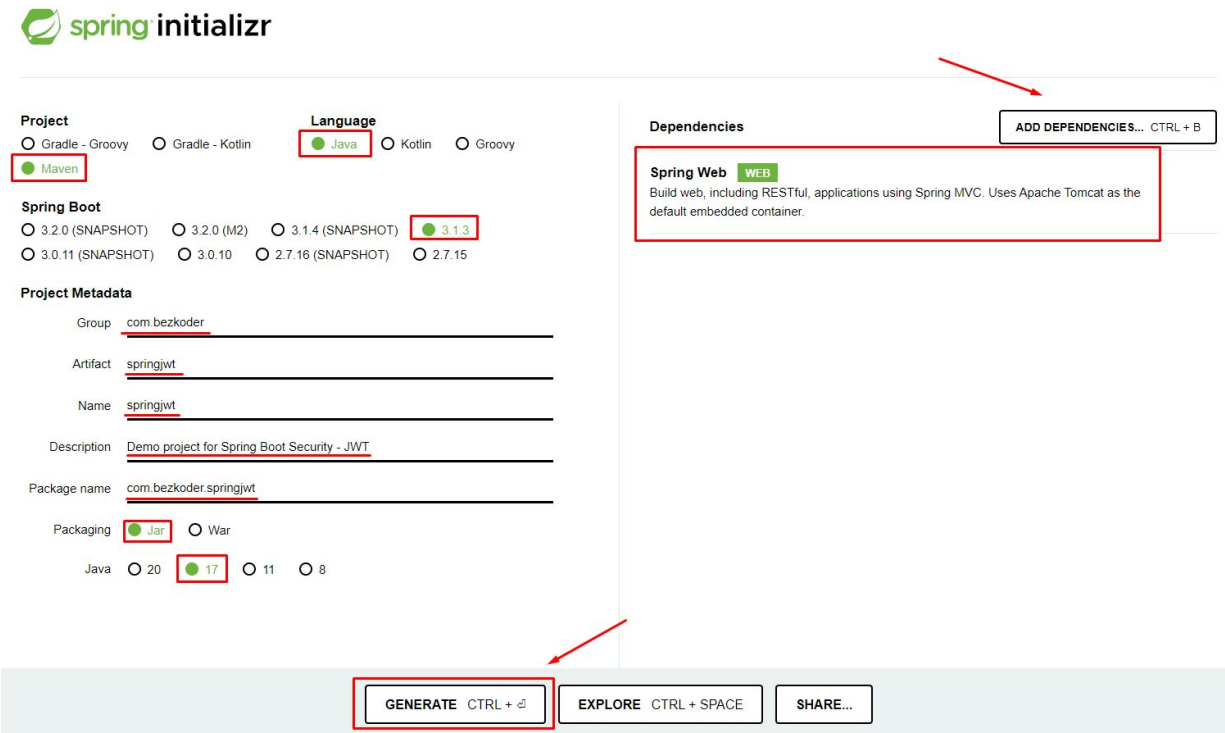
– `AuthController` handles signup/login requests

– `TestController` has accessing protected resource methods with role based validations.

Project Setup

Step 1: Create a New Spring Boot Project

- Go to https://start.spring.io/
- Choose:
  - Project: **Maven**
  - Language: **Java**
  - Spring Boot: **3.1.3**
- Project Metadata:
  - Group: *com.bezkoder*
  - Artifact: *springjwt*
  - Name: *springjwt*
  - Description**:** *Demo project for Spring Boot Security - JWT*
  - Package name: *com.bezkoder.springjwt*
  - Packaging: **Jar**
  - Java: **17**
- Add Dependencies: **Spring Web**
- Click "**GENERATE**"
- Open Project with IntelliJ



Add dependencies to pom.xml file

- Spring Boot
- Spring Data JPA
- Spring Security
- Spring Validation
- Spring Web
- MySQL Connector
- JSON Web Token (JWT) libraries
- testing libraries

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

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

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

<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

Step 2: Configure application.properties for MySQL

In your project's root directory, locate the `src/main/resources` folder. Right-click on it and select "New" > "File."
Name the file `application.properties`.
Open `application.properties` and configure it for MySQL. Add the following lines:

pom.xml (springjwt) | application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/your_database_name
spring.datasource.username=your_mysql_username
spring.datasource.password=your_mysql_password
spring.jpa.hibernate.ddl-auto=update
```

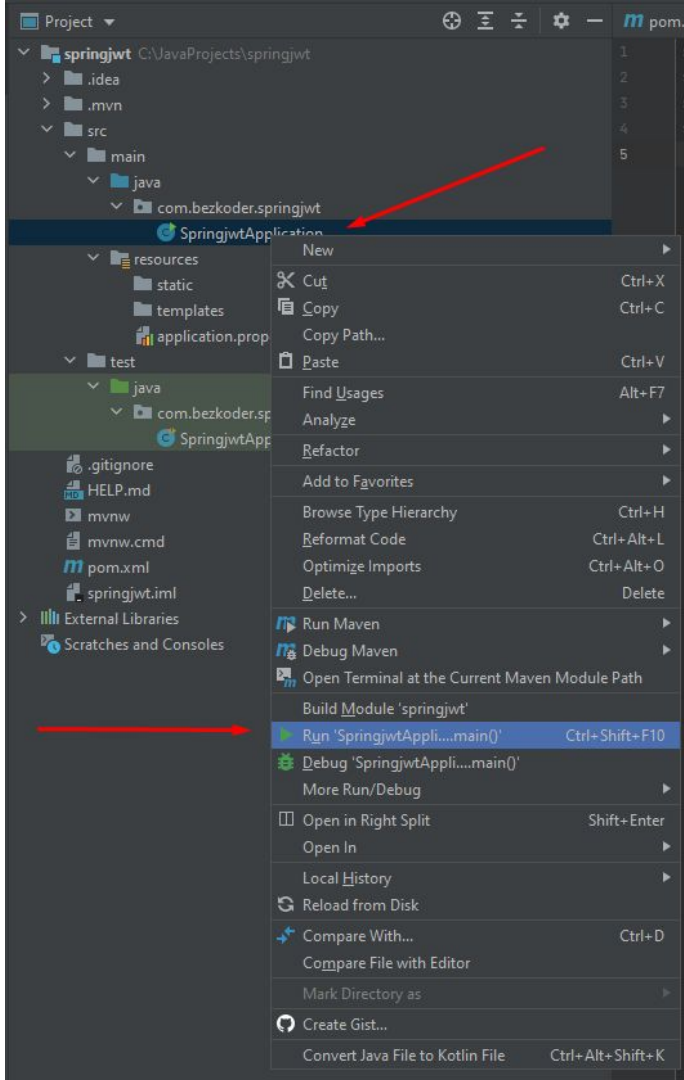
Replace `your_database_name`, `your_mysql_username`, and `your_mysql_password` with your actual MySQL database details.
Save the `application.properties` file.

Step 3: Build and Run Your Project

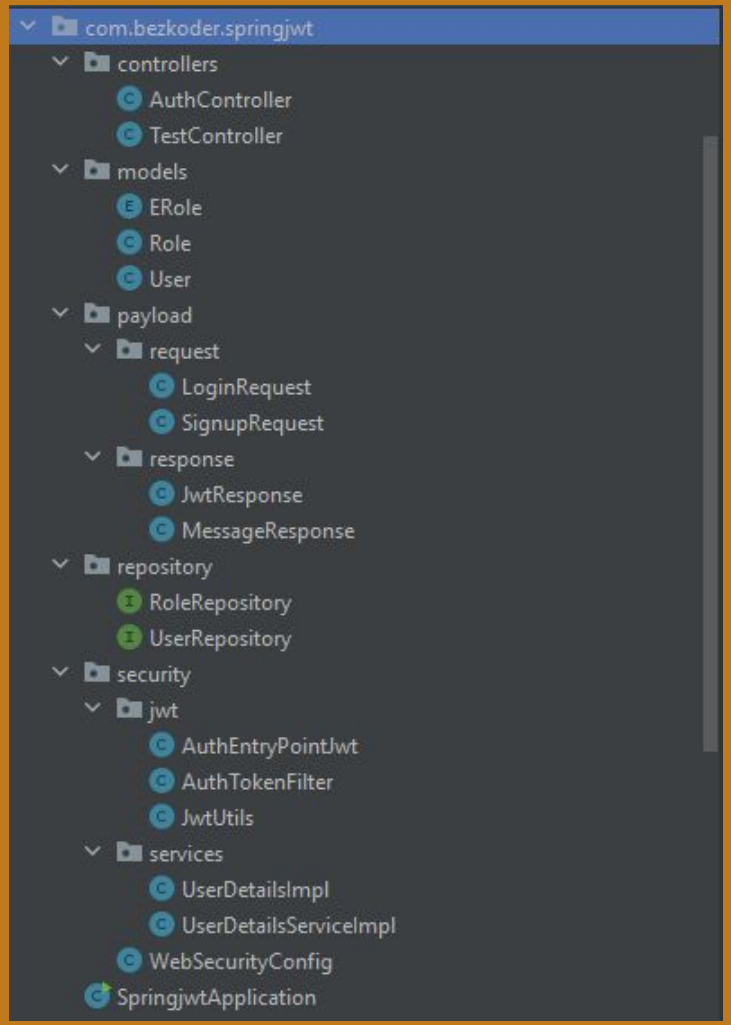
In IntelliJ IDEA, locate the main application class (`SpringjwtApplication.java`) in the `src/main/javacom/bezkoder/springjwt` directory.

Right-click on the main class and select "Run 'Application'."

IntelliJ IDEA will build your project and start the Spring Boot application. You should see output in the console indicating that the application has started successfully.



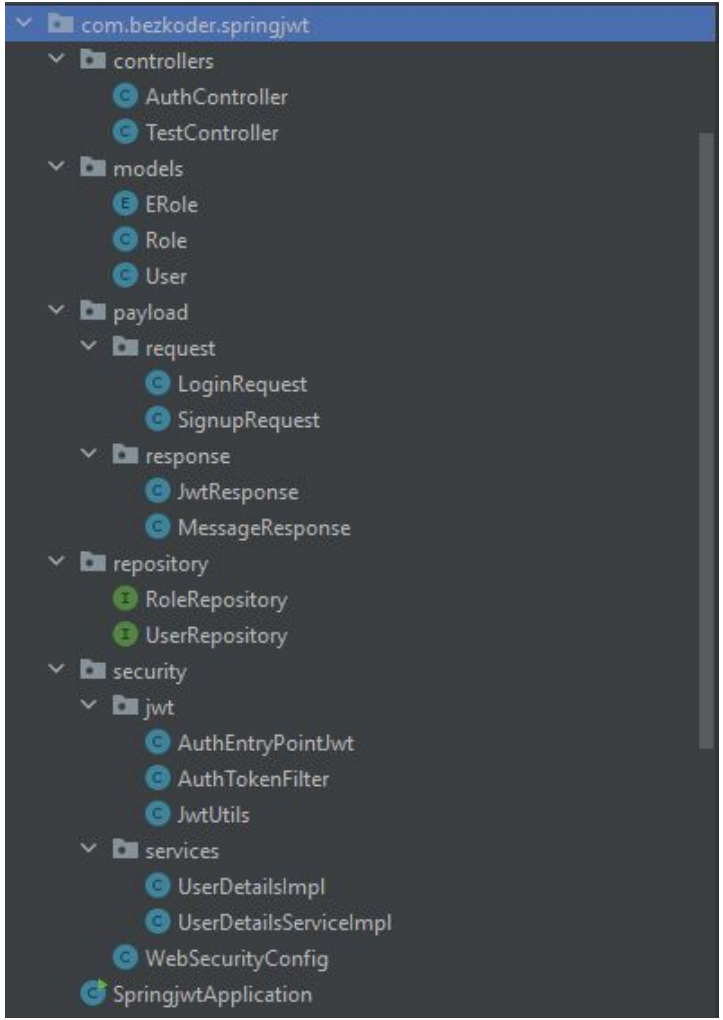
Project Structure



In the Spring Boot project structure typically follows best practices to maintain a clean and organized codebase. Below is a description of the common components and their purposes in the project structure

Controllers

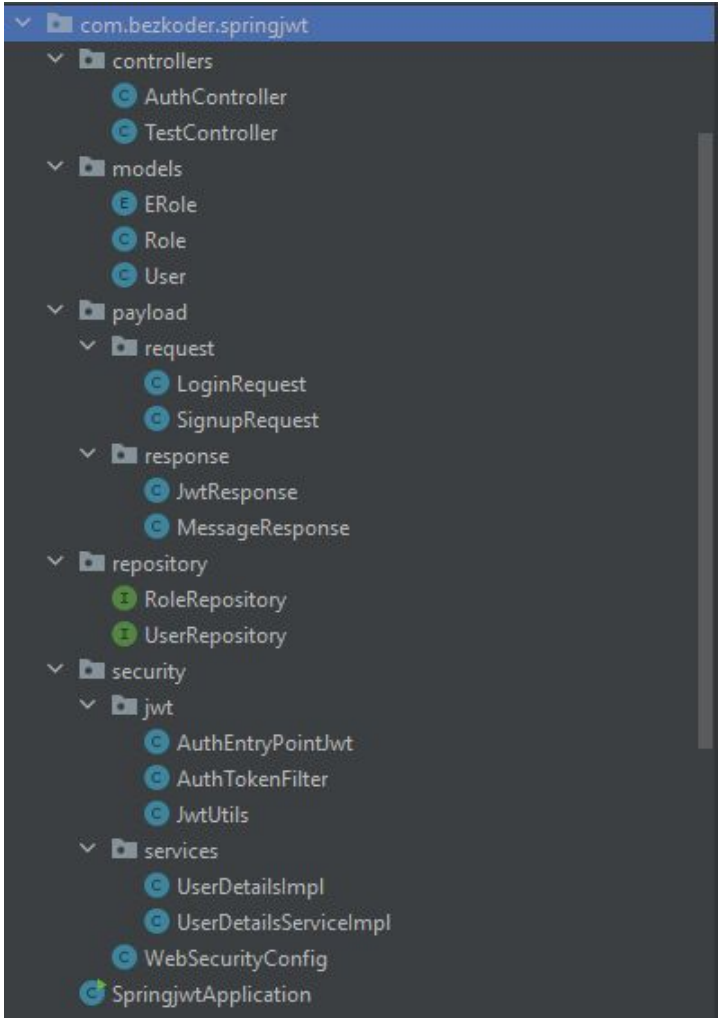
Controllers (AuthController and TestController) handle incoming HTTP requests. AuthController is mainly responsible for user authentication, while TestController manages endpoints with different levels of access.



Models

- `ERole` defines the possible roles that a user can have.
- `Role` represents these roles in the database and can be associated with users.
- `User` represents the user entity and contains a set of roles, establishing a many-to-many relationship with roles. This allows users to have one or more roles.

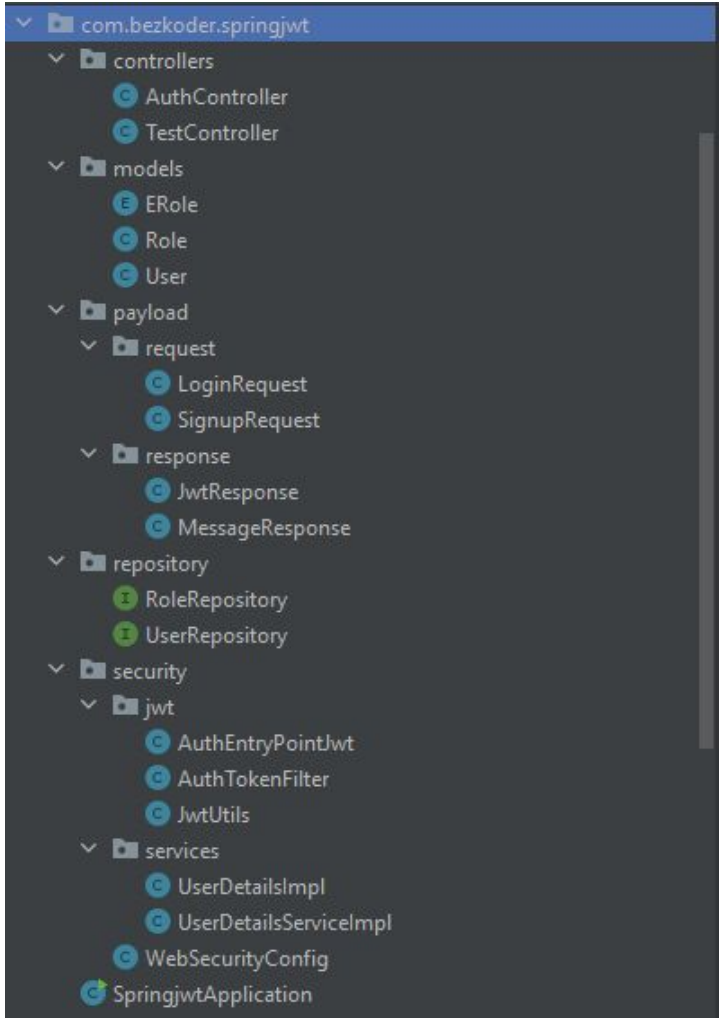
These classes collectively define the data model for user roles and users in the application. User roles can be assigned to users, and this relationship is managed through the `Role` and `User` classes.



Repository

- The `RoleRepository` and `UserRepository` interfaces are part of the data access layer of your application.
- These repositories provide methods to interact with the database without writing explicit SQL queries.
- The `RoleRepository` helps manage user roles, and the `UserRepository` manages user-related data.
- In your application's services or controllers, you can use these repositories to perform CRUD (Create, Read, Update, Delete) operations on roles and users.
- These repositories also enable you to query the database for specific roles or users based on certain criteria.

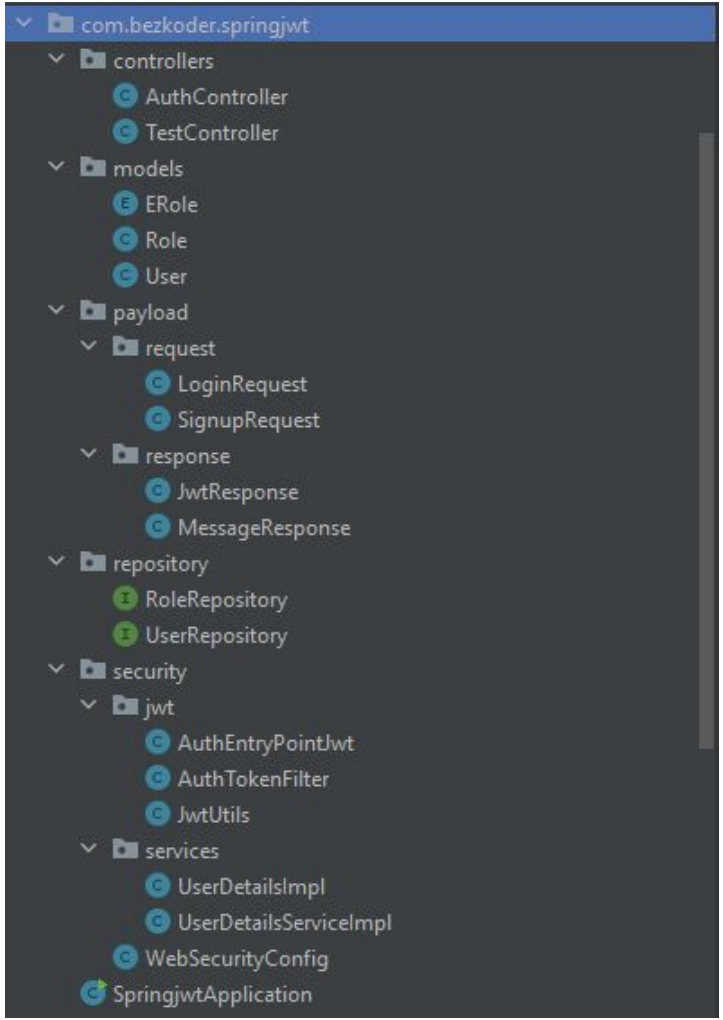
In summary, these repository interfaces are essential for abstracting and simplifying database operations within your Spring Boot application. They allow you to focus on your application's logic while providing a convenient way to interact with the underlying database tables for roles and users.



Security/jwt

- When a user attempts to access a protected resource, the `AuthTokenFilter` intercepts the request.
- It extracts the JWT token from the request's `Authorization` header and verifies its validity using the `JwtUtils`.
- If the token is valid, the user's authentication is set up in the `SecurityContextHolder`.
- If the token is invalid or missing, the `AuthEntryPointJwt` is invoked to send an unauthorized response.
- This combination of classes ensures that only authenticated users with valid JWT tokens can access protected resources, providing security to your Spring Boot application.

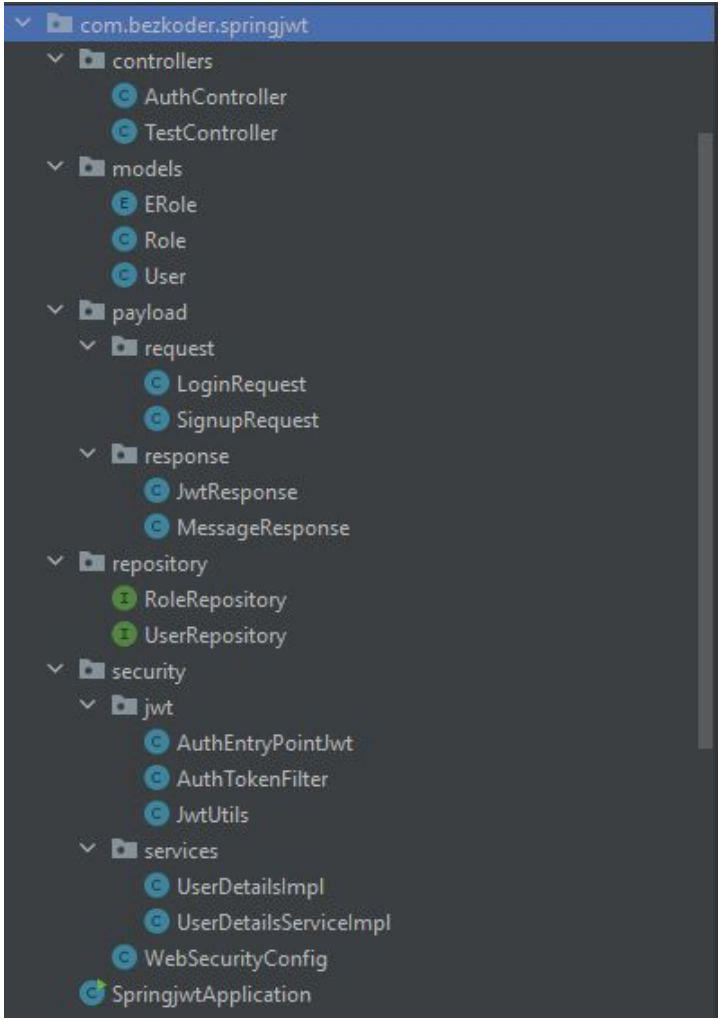
These JWT-related classes play a crucial role in securing your application by enabling token-based authentication and authorization. They work together to validate and process JWT tokens, ensuring that only authorized users can access protected endpoints.



Security/services

- When a user tries to authenticate, Spring Security's authentication manager calls the `loadUserByUsername` method of the `UserDetailsServiceImpl` class.
- The `UserDetailsServiceImpl` class fetches the user details from the data store using the `UserRepository`.
- The user details are then encapsulated in a `UserDetailsImpl` object.
- Spring Security uses this `UserDetailsImpl` object to perform authentication and authorization checks during the user's session.
- The `UserDetailsImpl` class provides user-specific details, including roles, which are used to determine whether the user is authorized to access certain resources (endpoints) in your application.

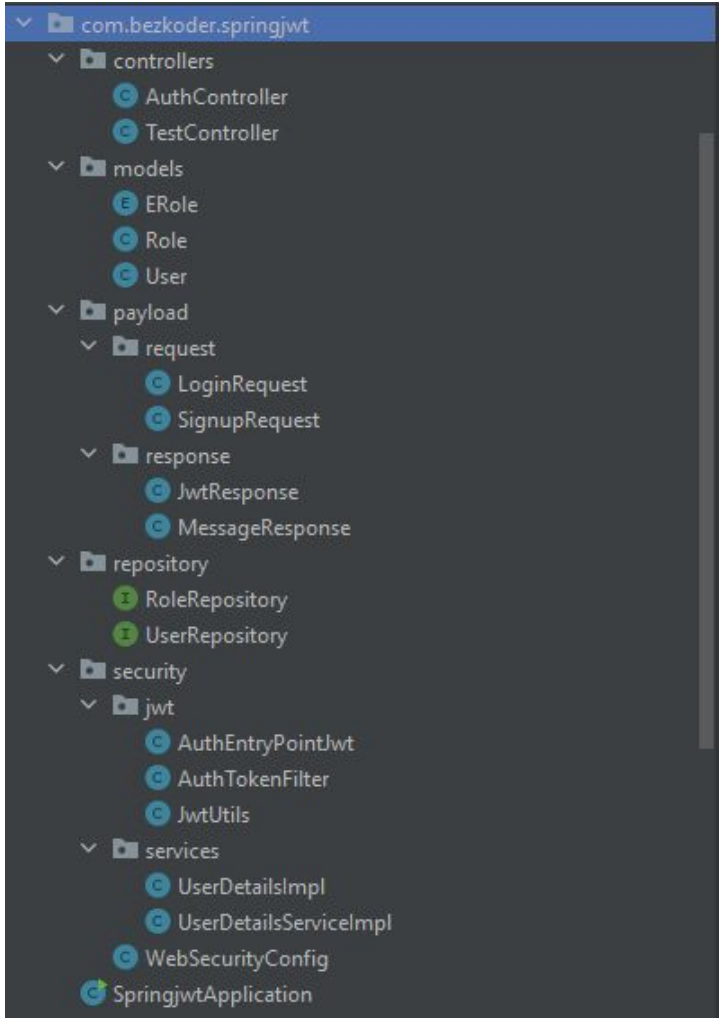
In summary, these classes work together to facilitate user authentication and authorization within your Spring Boot application. They retrieve user details from your data store and provide these details to Spring Security for authentication and authorization checks, ensuring that only authorized users can access protected resources.



WebSecurityConfig

- The `WebSecurityConfig` class plays a central role in configuring security settings within your Spring Boot application.
- It defines how authentication and authorization are handled, including the use of JWTs for authentication.
- The `AuthTokenFilter` intercepts requests and processes JWTs, and it is added to the filter chain in the `WebSecurityConfig` class.
- This configuration ensures that only authenticated users with valid JWTs can access protected resources, while some endpoints (like `/api/auth/**` and `/api/test/**`) are made accessible without authentication.

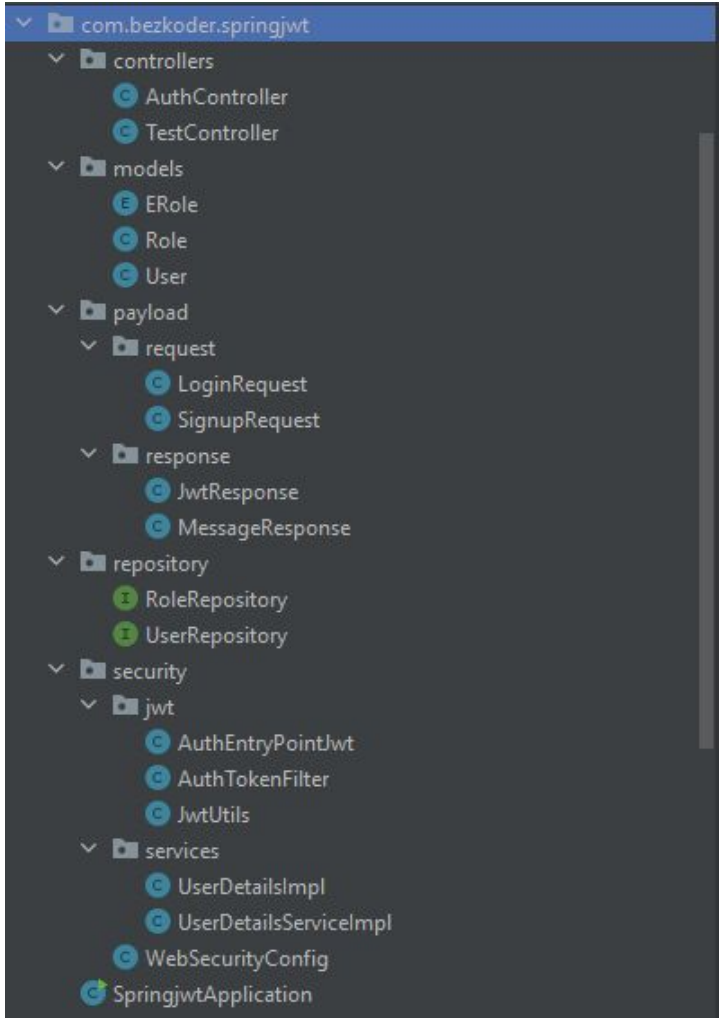
Overall, the `WebSecurityConfig` class is a critical component of your Spring Boot application, providing the necessary security configurations to protect your endpoints and manage user authentication.



Payload

Payload classes play a crucial role in our application's communication between the client and server. They help in structuring the data that is sent as part of HTTP requests and responses.

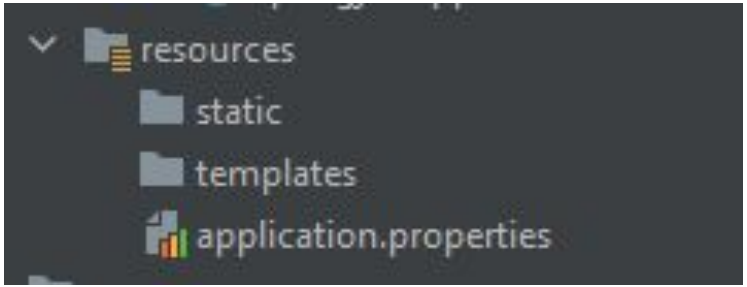
For example, `LoginRequest` captures login credentials from the client, and `JwtResponse` packages the authentication response with a JWT token. Similarly, `SignupRequest` captures user registration details, and `MessageResponse` is used to send simple textual messages back to the client. These classes ensure a consistent and well-defined format for data exchange between the client and server, which is essential for a well-structured RESTful API.



resources

src/main/resources: This is the main resource folder for your project.

- application.properties: Configuration properties for your Spring Boot application, including database connection details, server port, and other settings.
- static: Static resources like HTML, CSS, JavaScript, etc., if your project serves web pages.
- templates: If your project uses server-side rendering, HTML templates go here.



Create Entity Classes

Now, let's create the entity classes that represent your database tables.

Create Role.java

- This class is an entity class that represents user roles in the database.
- It is annotated with `@Entity` to indicate that it is a JPA entity and `@Table` specifies the name of the database table.
- The class includes fields for role ID, and an enumerated role name (`ERole`).
- The role name is defined as an enumerated type, which corresponds to the roles defined in `ERole.java`.
- This class is used to manage and store roles in the database.

```java
package com.bezkoder.springjwt.models;

import jakarta.persistence.*;

@Entity
@Table(name = "roles")
public class Role {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @Enumerated(EnumType.STRING)
  @Column(length = 20)
  private ERole name;

  public Role() {

  }

  public Role(ERole name) { this.name = name; }

  public Integer getId() { return id; }

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

  public ERole getName() { return name; }

  public void setName(ERole name) { this.name = name; }
}
```

Create User.java

- This class represents user entities in the application.
- It is also an entity class annotated with `@Entity`, and it defines the user table in the database.
- The class includes fields for user ID, username, email, password, and a set of roles.
- The `@NotBlank`, `@Size`, and `@Email` annotations define constraints on the user's username, email, and password.
- The `@ManyToMany` annotation establishes a many-to-many relationship between users and roles.
- Users can have multiple roles, and roles can belong to multiple users.
- The `JoinTable` annotation specifies the name of the join table that connects users and roles.

```java
package com.bezkoder.springjwt.models;

import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "users",
    uniqueConstraints = {
      @UniqueConstraint(columnNames = "username"),
      @UniqueConstraint(columnNames = "email")
    })
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @NotBlank
  @Size(max = 20)
  private String username;

  @NotBlank
  @Size(max = 50)
  @Email
  private String email;

  @NotBlank
  @Size(max = 120)
  private String password;

  @ManyToMany(fetch = FetchType.LAZY)
  @JoinTable(  name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id"))
  private Set<Role> roles = new HashSet<>();

  // Constructors, getters, and setters
```

Create User.java

Constructors, getters, and setters

```java
41    // Constructors, getters, and setters
42
43    public User() {
44    }
45
46    public User(String username, String email, String password) {
47      this.username = username;
48      this.email = email;
49      this.password = password;
50    }
51
52    public Long getId() { return id; }
55
56    public void setId(Long id) { this.id = id; }
59
60    public String getUsername() { return username; }
63
64    public void setUsername(String username) { this.username = username; }
67
68    public String getEmail() { return email; }
71
72    public void setEmail(String email) { this.email = email; }
75
76    public String getPassword() { return password; }
79
80    public void setPassword(String password) { this.password = password; }
83
84    public Set<Role> getRoles() { return roles; }
87
88    public void setRoles(Set<Role> roles) { this.roles = roles; }
91  }
92
```

ERole.java

- This class is an enumeration (enum) that represents different user roles.
- It defines three roles: ROLE_USER, ROLE_MODERATOR, and ROLE_ADMIN.

```java
package com.bezkoder.springjwt.models;

public enum ERole {
  ROLE_USER,
  ROLE_MODERATOR,
  ROLE_ADMIN
}
```

Create Payload Classes

Create LoginRequest.java

- This class represents a request payload for user login.
- It contains two fields: `username` and `password`, both annotated with `@NotBlank` to ensure they are not empty.
- The purpose of this class is to capture the login credentials (username and password) provided by the user during login.

```java
package com.bezkoder.springjwt.payload.request;

import jakarta.validation.constraints.NotBlank;

public class LoginRequest {
    @NotBlank
  private String username;

    @NotBlank
    private String password;

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }

    public void setPassword(String password) { this.password = password; }
}
```

Create SignupRequest.java

- This class represents a request payload for user registration (signup).
- It contains fields for `username`, `email`, `role`, and `password`, each with validation annotations.
  a. `@NotBlank` ensures that `username`, `email`, and `password` are not empty.
  b. `@Size` defines constraints on the length of `username` and `password`.
  c. `@Email` ensures that the `email` field follows an email format.
- The `role` field represents the user's role(s), and it's a set of strings.
- The purpose of this class is to capture the user's registration information, including username, email, role(s), and password.

```java
package com.bezkoder.springjwt.payload.request;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import java.util.Set;

public class SignupRequest {
  @NotBlank
  @Size(min = 3, max = 20)
  private String username;

  @NotBlank
  @Size(max = 50)
  @Email
  private String email;

  private Set<String> role;

  @NotBlank
  @Size(min = 6, max = 40)
  private String password;

  public String getUsername() { return username; }

  public void setUsername(String username) { this.username = username; }

  public String getEmail() { return email; }

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

  public String getPassword() { return password; }

  public void setPassword(String password) { this.password = password; }

  public Set<String> getRole() { return this.role; }

  public void setRole(Set<String> role) { this.role = role; }
}
```

Create JwtResponse.java

- This class represents a response payload for successful authentication.
- It contains fields for `token`, `type`, `id`, `username`, `email`, and `roles`.
- The `token` field holds the JWT (JSON Web Token) used for authentication.
- `type` indicates the token type, which is typically "Bearer" for JWT.
- `id`, `username`, `email`, and `roles` represent user information and roles associated with the authenticated user.
- The purpose of this class is to package the authentication response, including the JWT and user information.

```java
package com.bezkoder.springjwt.payload.response;

import java.util.List;

public class JwtResponse {
  private String token;
  private String type = "Bearer";
  private Long id;
  private String username;
  private String email;
  private List<String> roles;

  public JwtResponse(String accessToken, Long id, String username, String email, List<String> roles) {
    this.token = accessToken;
    this.id = id;
    this.username = username;
    this.email = email;
    this.roles = roles;
  }

  public String getAccessToken() { return token; }

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

  public String getTokenType() { return type; }

  public void setTokenType(String tokenType) { this.type = tokenType; }

  public Long getId() { return id; }

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

  public String getEmail() { return email; }

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

  public String getUsername() { return username; }

  public void setUsername(String username) { this.username = username; }

  public List<String> getRoles() { return roles; }
```

Create MessageResponse.java

- This class represents a generic response payload for sending messages.
- It contains a single field, `message`.
- The purpose of this class is to encapsulate simple messages in a response, which can be used for various purposes such as success messages or error messages.

```java
package com.bezkoder.springjwt.payload.response;

public class MessageResponse {
  private String message;

  public MessageResponse(String message) {
    this.message = message;
  }

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

Create Repository Interfaces

Create RoleRepository.java

- This repository interface extends `JpaRepository`, which is provided by Spring Data JPA.
- It is responsible for managing data related to user roles.
- The `Role` entity is the main class associated with this repository.
- The `Optional<Role> findByName(ERole name)` method allows you to find a role by its name (`ERole`). This is useful for looking up roles based on their names.

```java
package com.bezkoder.springjwt.repository;

import com.bezkoder.springjwt.models.ERole;
import com.bezkoder.springjwt.models.Role;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface RoleRepository extends JpaRepository<Role, Long> {
  Optional<Role> findByName(ERole name);
}
```

Create UserRepository.java

- Similar to the `RoleRepository`, this repository interface also extends `JpaRepository`.
- It manages data related to users in the application.
- The `User` entity is associated with this repository.
- The `Optional<User> findByUsername(String username)` method allows you to find a user by their username.
- The `Boolean existsByUsername(String username)` and `Boolean existsByEmail(String email)` methods are used for checking if a user with a specific username or email already exists in the database.

```java
package com.bezkoder.springjwt.repository;

import com.bezkoder.springjwt.models.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
  Optional<User> findByUsername(String username);

  Boolean existsByUsername(String username);

  Boolean existsByEmail(String email);
}
```

Create Security Classes

Create UserDetailsImpl

Create the `UserDetailsImpl` class that implements the `UserDetails` interface. This class is responsible for customizing the user details loaded by Spring Security. Ensure it has fields such as `id`, `username`, `email`, `password`, and `authorities`.

```java
package com.bezkoder.springjwt.security.services;

import com.bezkoder.springjwt.models.User;
import com.fasterxml.jackson.annotation.JsonIgnore;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import java.util.Collection;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

public class UserDetailsImpl implements UserDetails {
  private static final long serialVersionUID = 1L;
  private Long id;
  private String username;
  private String email;
  @JsonIgnore
  private String password;
  private Collection<? extends GrantedAuthority> authorities;

  public UserDetailsImpl(Long id, String username, String email, String password,
      Collection<? extends GrantedAuthority> authorities) {
    this.id = id;
    this.username = username;
    this.email = email;
    this.password = password;
    this.authorities = authorities;
  }

  public static UserDetailsImpl build(User user) {
    List<GrantedAuthority> authorities = user.getRoles().stream()
        .map(role -> new SimpleGrantedAuthority(role.getName().name()))
        .collect(Collectors.toList());

    return new UserDetailsImpl(
        user.getId(),
        user.getUsername(),
        user.getEmail(),
        user.getPassword(),
        authorities);
  }
```

Continued on next slide

Create UserDetailsImpl

```java
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() { return authorities; }

    public Long getId() { return id; }

    public String getEmail() { return email; }

    @Override
    public String getPassword() { return password; }

    @Override
    public String getUsername() { return username; }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }

    @Override
    public boolean equals(Object o) {
      if (this == o)
        return true;
      if (o == null || getClass() != o.getClass())
        return false;
      UserDetailsImpl user = (UserDetailsImpl) o;
      return Objects.equals(id, user.id);
    }
  }
```

Create UserDetailsServiceImpl

Implement the `UserDetailsService` interface in the `UserDetailsServiceImpl` class. This service loads user details from your database using a `UserRepository`. The `loadUserByUsername` method should retrieve user information and construct a `UserDetailsImpl` object.

```java
package com.bezkoder.springjwt.security.services;

import com.bezkoder.springjwt.models.User;
import com.bezkoder.springjwt.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {
  @Autowired
  UserRepository userRepository;

  @Override
  @Transactional
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    User user = userRepository.findByUsername(username)
        .orElseThrow(() -> new UsernameNotFoundException("User Not Found with username: " + username));

    return UserDetailsImpl.build(user);
  }

}
```

Create JwtUtils

Develop the `JwtUtils` class to handle JWT token generation, parsing, and validation. This class should include methods like `generateJwtToken`, `getUserNameFromJwtToken`, and `validateJwtToken`.

```java
package com.bezkoder.springjwt.security.jwt;

import com.bezkoder.springjwt.security.services.UserDetailsImpl;
import io.jsonwebtoken.*;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;

@Component
public class JwtUtils {
  private static final Logger logger = LoggerFactory.getLogger(JwtUtils.class);

  @Value("${bezkoder.app.jwtSecret}")
  private String jwtSecret;

  @Value("${bezkoder.app.jwtExpirationMs}")
  private int jwtExpirationMs;

  public String generateJwtToken(Authentication authentication) {

    UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication.getPrincipal();

    return Jwts.builder()
        .setSubject((userPrincipal.getUsername()))
        .setIssuedAt(new Date())
        .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))
        .signWith(key(), SignatureAlgorithm.HS256)
        .compact();
  }
```

Continued on next slide

Create JwtUtils

```java
@ private Key key() {
    return Keys.hmacShaKeyFor(Decoders.BASE64.decode(jwtSecret));
  }

  public String getUserNameFromJwtToken(String token) {
    return Jwts.parserBuilder().setSigningKey(key()).build()
              .parseClaimsJws(token).getBody().getSubject();
  }

  public boolean validateJwtToken(String authToken) {
    try {
      Jwts.parserBuilder().setSigningKey(key()).build().parse(authToken);
      return true;
    } catch (MalformedJwtException e) {
      logger.error("Invalid JWT token: {}", e.getMessage());
    } catch (ExpiredJwtException e) {
      logger.error("JWT token is expired: {}", e.getMessage());
    } catch (UnsupportedJwtException e) {
      logger.error("JWT token is unsupported: {}", e.getMessage());
    } catch (IllegalArgumentException e) {
      logger.error("JWT claims string is empty: {}", e.getMessage());
    }

    return false;
  }
}
```

Create AuthTokenFilter

Build the `AuthTokenFilter` class by extending `OncePerRequestFilter`. This filter intercepts incoming requests, extracts JWT tokens from the request headers, validates them using `JwtUtils`, and sets the user's authentication in the `SecurityContextHolder` if the token is valid.

```java
package com.bezkoder.springjwt.security.jwt;

import com.bezkoder.springjwt.security.services.UserDetailsServiceImpl;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.util.StringUtils;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

public class AuthTokenFilter extends OncePerRequestFilter {
  @Autowired
  private JwtUtils jwtUtils;

  @Autowired
  private UserDetailsServiceImpl userDetailsService;

  private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);

```

Continued on next slide

Create AuthTokenFilter

```java
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
      try {
        String jwt = parseJwt(request);
        if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
          String username = jwtUtils.getUserNameFromJwtToken(jwt);

          UserDetails userDetails = userDetailsService.loadUserByUsername(username);
          UsernamePasswordAuthenticationToken authentication =
              new UsernamePasswordAuthenticationToken(
                  userDetails,
                  credentials: null,
                  userDetails.getAuthorities());
          authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

          SecurityContextHolder.getContext().setAuthentication(authentication);
        }
      } catch (Exception e) {
        logger.error("Cannot set user authentication: {}", e);
      }

      filterChain.doFilter(request, response);
    }

    private String parseJwt(HttpServletRequest request) {
      String headerAuth = request.getHeader( s: "Authorization");

      if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer ")) {
        return headerAuth.substring(7);
      }

      return null;
    }
  }
```

Create AuthEntryPointJwt

Develop the `AuthEntryPointJwt` class, implementing the `AuthenticationEntryPoint` interface. This class handles unauthorized access attempts by returning an appropriate JSON response.

```java
package com.bezkoder.springjwt.security.jwt;

import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.MediaType;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

  private static final Logger logger = LoggerFactory.getLogger(AuthEntryPointJwt.class);

  @Override
  public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)
      throws IOException, ServletException {
    logger.error("Unauthorized error: {}", authException.getMessage());

    response.setContentType(MediaType.APPLICATION_JSON_VALUE);
    response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

    final Map<String, Object> body = new HashMap<>();
    body.put("status", HttpServletResponse.SC_UNAUTHORIZED);
    body.put("error", "Unauthorized");
    body.put("message", authException.getMessage());
    body.put("path", request.getServletPath());

    final ObjectMapper mapper = new ObjectMapper();
    mapper.writeValue(response.getOutputStream(), body);
  }

}
```

Create WebSecurityConfig

Create the `WebSecurityConfig` class as your main security configuration. This class configures security rules, authentication providers, and filters. It should include:

- Configuration annotations (`@Configuration`, `@EnableMethodSecurity`, etc.).
- `@Autowired` fields for `UserDetailsServiceImpl` and `AuthEntryPointJwt`.
- Bean definitions for `AuthTokenFilter`, `DaoAuthenticationProvider`, `AuthenticationManager`, and `PasswordEncoder`.
- Configuration for the security filter chain using `.csrf()`, `.exceptionHandling()`, `.sessionManagement()`, and `.authorizeHttpRequests()`.
- Registration of the `AuthTokenFilter` before `UsernamePasswordAuthenticationFilter`.

Continued on next slide

```java
package com.bezkoder.springjwt.security;

import com.bezkoder.springjwt.security.jwt.AuthEntryPointJwt;
import com.bezkoder.springjwt.security.jwt.AuthTokenFilter;
import com.bezkoder.springjwt.security.services.UserDetailsServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableMethodSecurity

public class WebSecurityConfig { // extends WebSecurityConfigurerAdapter {
  @Autowired
  UserDetailsServiceImpl userDetailsService;

  @Autowired
  private AuthEntryPointJwt unauthorizedHandler;

  @Bean
  public AuthTokenFilter authenticationJwtTokenFilter() { return new AuthTokenFilter(); }

  @Bean
  public DaoAuthenticationProvider authenticationProvider() {
      DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

      authProvider.setUserDetailsService(userDetailsService);
      authProvider.setPasswordEncoder(passwordEncoder());

      return authProvider;
  }
```

Create WebSecurityConfig

```java
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig) throws Exception {
      return authConfig.getAuthenticationManager();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http.csrf(csrf -> csrf.disable())
          .exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))
          .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
          .authorizeHttpRequests(auth ->
            auth.requestMatchers( ...patterns: "/api/auth/**").permitAll()
                .requestMatchers( ...patterns: "/api/test/**").permitAll()
                .anyRequest().authenticated()
          );

      http.authenticationProvider(authenticationProvider());

      http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

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

Create Spring RestAPIs Controllers

Controller for Authentication

This controller provides APIs for register and login actions.

– `/api/auth/signin`

- authenticate { username, pasword }
- update `SecurityContext` using `Authentication` object
- generate `JWT`
- get `UserDetails` from `Authentication` object
- response contains `JWT` and `UserDetails` data

Continued on next slide

```java
package com.bezkoder.springjwt.controllers;

import ...

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/auth")
public class AuthController {
  @Autowired
  AuthenticationManager authenticationManager;

  @Autowired
  UserRepository userRepository;
  @Autowired
  RoleRepository roleRepository;
  @Autowired
  PasswordEncoder encoder;
  @Autowired
  JwtUtils jwtUtils;

  @PostMapping("/signin")
  public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest) {

    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));

    SecurityContextHolder.getContext().setAuthentication(authentication);
    String jwt = jwtUtils.generateJwtToken(authentication);

    UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
    List<String> roles = userDetails.getAuthorities().stream()
        .map(item -> item.getAuthority())
        .collect(Collectors.toList());

    return ResponseEntity.ok(new JwtResponse(jwt,
                         userDetails.getId(),
                         userDetails.getUsername(),
                         userDetails.getEmail(),
                         roles));
  }
```

Controller for Authentication

This controller provides APIs for register and login actions.

– `/api/auth/signup`

- check existing `username`/`email`
- create new `User` (with `ROLE_USER` if not specifying role)
- save `User` to database using `UserRepository`

```java
@PostMapping("/signup")
public ResponseEntity<?> registerUser(@Valid @RequestBody SignupRequest signUpRequest) {
  if (userRepository.existsByUsername(signUpRequest.getUsername())) {
    return ResponseEntity
        .badRequest()
        .body(new MessageResponse("Error: Username is already taken!"));
  }
  if (userRepository.existsByEmail(signUpRequest.getEmail())) {
    return ResponseEntity
        .badRequest()
        .body(new MessageResponse("Error: Email is already in use!"));
  }
  // Create new user's account
  User user = new User(signUpRequest.getUsername(),
             signUpRequest.getEmail(),
             encoder.encode(signUpRequest.getPassword()));
  Set<String> strRoles = signUpRequest.getRole();
  Set<Role> roles = new HashSet<>();
  if (strRoles == null) {
    Role userRole = roleRepository.findByName(ERole.ROLE_USER)
        .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
    roles.add(userRole);
  } else {
    strRoles.forEach(role -> {
      switch (role) {
      case "admin":
        Role adminRole = roleRepository.findByName(ERole.ROLE_ADMIN)
            .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
        roles.add(adminRole);
        break;
      case "mod":
        Role modRole = roleRepository.findByName(ERole.ROLE_MODERATOR)
            .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
        roles.add(modRole);
        break;
      default:
        Role userRole = roleRepository.findByName(ERole.ROLE_USER)
            .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
        roles.add(userRole);
      }
    });
  }
  user.setRoles(roles);
  userRepository.save(user);
  return ResponseEntity.ok(new MessageResponse("User registered successfully!"));
}
}
```

Controller for testing Authorization

There are 4 APIs:

– `/api/test/all` for public access

– `/api/test/user` for users has `ROLE_USER` or `ROLE_MODERATOR` or `ROLE_ADMIN`

– `/api/test/mod` for users has `ROLE_MODERATOR`

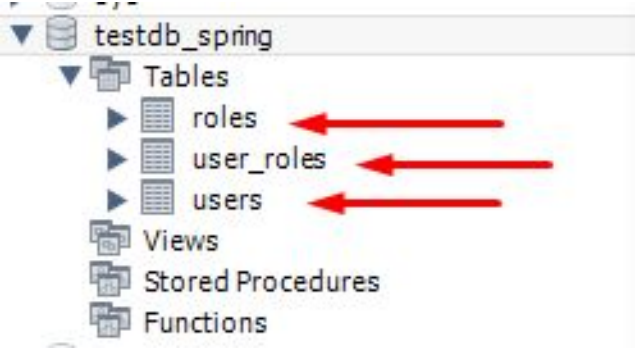
– `/api/test/admin` for users has `ROLE_ADMIN`

```java
package com.bezkoder.springjwt.controllers;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/test")
public class TestController {
  @GetMapping("/all")
  public String allAccess() {
    return "Public Content.";
  }

  @GetMapping("/user")
  @PreAuthorize("hasRole('USER') or hasRole('MODERATOR') or hasRole('ADMIN')")
  public String userAccess() {
    return "User Content.";
  }

  @GetMapping("/mod")
  @PreAuthorize("hasRole('MODERATOR')")
  public String moderatorAccess() {
    return "Moderator Board.";
  }

  @GetMapping("/admin")
  @PreAuthorize("hasRole('ADMIN')")
  public String adminAccess() {
    return "Admin Board.";
  }
}
```

Run & Test

Run Project

- Run Spring Boot application
- Tables that we define in *models* package will be automatically generated in Database.



Add Roles to DB

We also need to add some rows into roles table before assigning any role to User.

Run following SQL insert statements

```
INSERT INTO testdb_spring.roles(name) VALUES('ROLE_USER');
INSERT INTO testdb_spring.roles(name) VALUES('ROLE_MODERATOR');
INSERT INTO testdb_spring.roles(name) VALUES('ROLE_ADMIN');
```

Then check the tables:

```
SELECT * FROM testdb_spring.roles;
```

Result Grid | Filter Rows:

id	name
1	ROLE_USER
2	ROLE_MODERATOR
3	ROLE_ADMIN
NULL	NULL

Test for /signup API Request

Register some users with `/signup` API:

POST http://localhost:8080/api/auth/signup Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2     "username": "moderator",
3     "email": "moderator@gmail.com",
4     "password": "123456",
5     "role": ["mod", "user"]
6 }
```

Body Cookies Headers (14) Test Results Status: 200 OK Time: 698 ms Size: 466 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2     "message": "User registered successfully!"
3 }
```

Access public resource: GET `/api/test/all`

securitySpring / **Access public resource**

Save

GET http://localhost:8080/api/test/all **Send**

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

	KEY	VALUE	DESCRIPTION	ooo	Bulk Edit
	Key	Value	Description		

Body Cookies Headers (14) Test Results Status: 200 OK Time: 61 ms Size: 438 B Save Response

Pretty Raw Preview Visualize Text

```
1  Public Content.
```

Access protected resource: GET `/api/test/user`

securitySpring / Access protected resource

Save

GET http://localhost:8080/api/test/user Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

Body Cookies Headers (14) Test Results Status: 401 Unauthorized Time: 91 ms Size: 555 B Save Response

Pretty Raw Preview Visualize JSON


```
{
    "path": "/api/test/user",
    "error": "Unauthorized",
    "message": "Full authentication is required to access this resource",
    "status": 401
}
```

Login an account: POST `/api/auth/signin`

securitySpring / Login an account

Save

POST http://localhost:8080/api/auth/signin Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2     "username": "moderator",
3     "password": "123456"
4 }
```

Body Cookies Headers (14) Test Results Status: 200 OK Time: 269 ms Size: 697 B Save Response

Pretty Raw Preview Visualize JSON


```
1 {
2     "id": 1,
3     "username": "moderator",
4     "email": "moderator@gmail.com",
5     "roles": [
6         "ROLE_MODERATOR",
7         "ROLE_USER"
8     ],
9     "accessToken": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJtb2RlcmF0b3IiLCJpYXQiOjE2OTM5ODE1NTQsImV4cCI6MTY5NDA2Nzk1NH0.
          kGPYgxp-6BOCJqP8zmCCfDRje3CNtfyFELNirMRXl14",
10    "tokenType": "Bearer"
11 }
```

Access `ROLE_USER` resource: GET `/api/test/user`

securitySpring / Access ROLE_USER resource

Save

GET http://localhost:8080/api/test/user Send

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings | Cookies

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJtb2RlcmI...

Set Token

Body | Cookies | Headers (14) | Test Results

Status: 200 OK Time: 60 ms Size: 436 B Save Response

Pretty | Raw | Preview | Visualize | Text

```
1  User Content.
```

Access ROLE_MODERATOR resource: GET /api/test/mod

securitySpring / Access ROLE_MODERATOR resource

Save

GET http://localhost:8080/api/test/mod Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Set Token

Token eyJhbGciOiJIUzI1NiJ9.eyJzdWliOiJtb2Rlcml...

Body Cookies Headers (14) Test Results Status: 200 OK Time: 25 ms Size: 439 B Save Response

Pretty Raw Preview Visualize Text

```
1  Moderator Board.
```

Register Admin with `/signup` API

securitySpring / SignUp Admin

Save

POST http://localhost:8080/api/auth/signup Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2     "username": "admin",
3     "email": "admin@gmail.com",
4     "password": "123456",
5     "role": ["admin"]
6 }
```

Body Cookies Headers (14) Test Results Status: 200 OK Time: 95 ms Size: 466 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2     "message": "User registered successfully!"
3 }
```

Login an admin account: POST `/api/auth/signin`

securitySpring / Login an admin account

Save

POST http://localhost:8080/api/auth/signin Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "username": "admin",
    "password": "123456"
}
```

Body Cookies Headers (14) Test Results Status: 200 OK Time: 78 ms Size: 668 B Save Response

Pretty Raw Preview Visualize JSON


```
{
    "id": 2,
    "username": "admin",
    "email": "admin@gmail.com",
    "roles": [
        "ROLE_ADMIN"
    ],
    "accessToken": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImlhdCI6MTY5Mzk4MjM0NiwiZXhwIjoxNjk0MDY4NzQ2fQ.
        VQPKvYIcby65eSH0wVTzAIyVNrm78f3Xvv3qItpWOaY",
    "tokenType": "Bearer"
}
```

Access ROLE_MODERATOR resource: GET /api/test/mod

securitySpring / Access ROLE_ADMIN resource

Save

GET http://localhost:8080/api/test/admin Send

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings | Cookies

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbil ...

Set Token

Body | Cookies | Headers (14) | Test Results

Status: 200 OK Time: 19 ms Size: 435 B Save Response

Pretty | Raw | Preview | Visualize | Text

```
1  Admin Board.
```

Conclusion

Conclusion

In the course of this project, we embarked on the development of a Spring Boot application with a focus on user authentication and authorization using JSON Web Tokens (JWT). This project has been an exploration into modern web security practices and has yielded several notable achievements and outcomes:

- **Authentication and Authorization**: We successfully implemented a robust authentication and authorization system. Users can securely register, log in, and access protected resources based on their assigned roles.
- **Role-based Access Control:** The project incorporated role-based access control, distinguishing between `ROLE_USER`, `ROLE_MODERATOR`, and `ROLE_ADMIN`. Each role comes with specific permissions, ensuring that users only access functionalities relevant to their roles.
- **Token-Based Authentication:** JSON Web Tokens (JWT) were utilized for authentication. JWTs provide a stateless and secure method of verifying user identity without the need for server-side sessions. This approach enhances scalability and security.
- **Spring Security**: We leveraged Spring Security, a powerful framework for securing Spring applications. It seamlessly integrated with our project, providing a high level of security with minimal configuration.
- **Database Integration:** Our application effectively connects to a MySQL database, storing user information, roles, and other critical data securely.
- **Exception Handling:** Proper error handling and custom exception classes were implemented to ensure a user-friendly and secure experience.
- Logging: Logging mechanisms were put in place, allowing for efficient debugging and monitoring of the application.

Conclusion

This project serves as a practical demonstration of the power of Spring Boot and modern security practices in building robust and secure web applications. It lays the foundation for more complex systems and can be extended with additional features and functionality as needed.

In conclusion, this project represents a significant step forward in mastering Spring Boot and JWT-based security. It offers a solid starting point for further development and serves as an excellent reference for anyone seeking to implement user authentication and authorization in their web applications.