M. Nardone, C. Scarioni, *Pro Spring Security*
https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_9

9. JSON Web Token (JWT) Authentication

Massimo Nardone[1] and Carlo Scarioni[2]
(1) HELSINKI, Finland
(2) Surbiton, UK

This chapter explores REST API and JWT authentication and authorization using Spring Boot 3 and Spring Security 6.

In previous chapters, you saw some types of Spring Security authentication methods. First, let's look at the REST API and an example of JWT authentication.

The REST API

REST, which stands for *REpresentational State Transfer*, is an architectural style for designing networked applications. REST has become the predominant way of designing an API (application programming interface) for web-based applications.

REST APIs provide a structured and standardized way for different software applications to communicate over the Internet. They've become the backbone of modern web and mobile applications, enabling seamless integration and interaction between various services and systems.

REST APIs allow different software applications to communicate and interact with each other over the Internet using standard HTTP methods.

The following describes REST API key concepts.

- **Resources**: In REST, everything is considered a resource. A resource can be a data entity, an object, or any other type of information that can be identified by a unique URL (Uniform Resource Locator).
- **HTTP methods**: REST APIs utilize standard HTTP methods to perform operations on resources like the following.
  - **GET** retrieves data from the server.
  - **POST** sends data to the server to create a new resource.
  - **PUT** updates an existing resource on the server.
  - **DELETE** removes a resource from the server.
  - **PATCH** partially updates a resource.
- **Uniform interface**: REST APIs have a uniform and consistent interface. Each resource is identified by a URL, and different HTTP methods are used to interact with those resources.
- **Stateless**: Each API request from a client to a server must contain all the information needed to understand and fulfill the request. The

server doesn't store any client state between requests.

- **Client-server architecture**: REST separates the client (the application making the request) from the server (the application fulfilling the request), which allows them to evolve independently.
- **Response format**: REST APIs typically return data in common formats such as JSON (JavaScript Object Notation) or XML (eXtensible Markup Language).

The following are simple examples of a REST API managing a list of cars.

- GET /cars: Retrieve a list of all cars.
- GET /cars/{id}: Retrieve details of a specific car.
- POST /cars: Create a new car record by sending car data.
- PUT /cars/{id}: Update details of a specific car.
- DELETE /cars/{id}: Delete a specific car.

The following describes the most important advantages of REST API.

- **Scalability**: RESTful architectures are scalable due to their stateless nature.
- **Flexibility**: Clients and servers can evolve independently without affecting each other if the API contract remains consistent.
- **Wide adoption**: REST is widely adopted and understood, making it easier for developers to work with.
- **Caching**: REST APIs can use HTTP caching mechanisms to improve performance.
- **Language and platform independence**: Since REST APIs use standard HTTP methods and formats, they can be accessed from various programming languages and platforms.

While REST APIs have numerous advantages, there are also some disadvantages and limitations to consider.

- **Lack of standardization**: Despite being a widely adopted architectural style, REST doesn't provide strict guidelines on how to design APIs. This can lead to inconsistencies in API design and make it challenging to ensure uniformity across different APIs.
- **Overfetching and underfetching**: REST APIs often return fixed data structures, which can lead to overfetching (receiving more data than needed) or underfetching (receiving less data than needed) of information. This can result in wasted bandwidth or additional requests.
- **Limited support for real-time communication**: REST APIs are typically request-response-based and may not be well-suited for real-time communication. Implementing features like instant messaging or live updates can be complex and might require additional technologies.
- **No built-in state management**: REST APIs are stateless, which means the server doesn't store the client state. While this simplifies server design, managing session-related information can lead to challenges.
- **Lack of rich semantics**: REST APIs primarily rely on HTTP methods and status codes, which may not always convey rich semantics about the underlying operations. This can lead to ambiguity in understanding the purpose of certain API endpoints.

- **Performance overhead**: REST APIs may involve additional data parsing and serialization steps due to reliance on formats like JSON or XML. This can introduce performance overhead, especially in high-frequency scenarios.
- **Multiple requests for complex operations**: Complex operations often require multiple requests to the server, leading to additional network overhead and latency. This can be a concern for mobile applications or in situations with limited bandwidth.
- **Lack of flexibility in versioning**: Changing a REST API while maintaining backward compatibility can be challenging. Different versions of the API might need to be managed, which can complicate the development and deployment process.
- **Security considerations**: While REST APIs can be secured using mechanisms like HTTPS and authentication, designing a secure REST API requires careful consideration of authorization, token management, and protection against common security vulnerabilities.
- **Limited discoverability**: Discovering the available endpoints and their functionalities in a REST API might require external documentation, as there’s no built-in mechanism for exposing the API structure to clients.

Introduction to JSON Web Token

JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JWTs are commonly used for authentication and authorization in web applications and APIs.

A JWT consists of three parts.

- **Header**: The header typically consists of two parts: the type of the token (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- **Payload**: The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically the user) and additional metadata and can be categorized into three types.
  - **Registered Claims** are predefined claims with specific meanings, like iss (issuer), exp (expiration time), sub (subject), and more.
  - **Public Claims** are custom claims that you define to convey additional information.
  - **Private Claims** are custom claims meant to be shared between parties that agree on their usage and are not defined in any public specification.

```
{

  "sub": "1234567890",

  "name": "Massimo Nardone",

  "iat": 6723561290

}
```

- **Signature**: To create a signature, you must sign the encoded header, the encoded payload, a secret, and the algorithm specified in the header. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
  The following explains how JWTs work.

- **Authentication**: When a user logs in, the server creates a JWT containing the user's information and signs it with a secret key. This JWT is then sent to the client.
- **Authorization**: The client includes the JWT in the headers of subsequent requests to the server. The server can then verify the JWT's signature and extract the user's information from the payload to grant access to protected resources.

Figure 9-1 shows how JWT works.

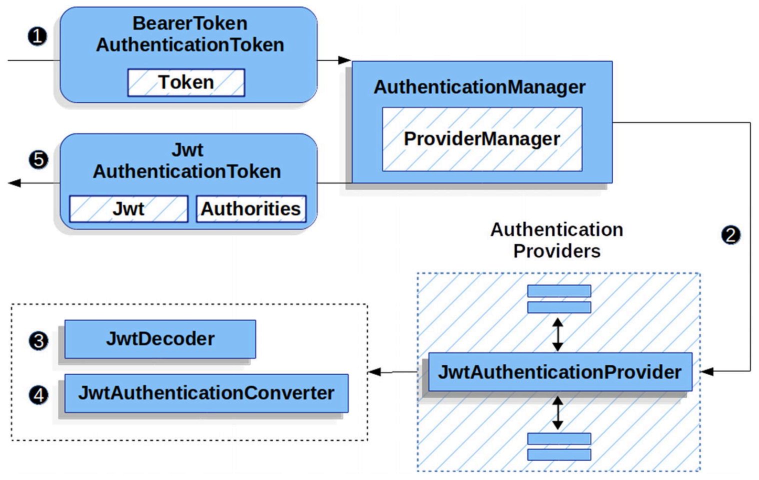


Figure 9-1 JWT working diagram (source docs.spring.io)

1. The authentication filter from reading the bearer token passes a BearerTokenAuthenticationToken to the AuthenticationManager, which is implemented by ProviderManager.
2. The ProviderManager is configured to use an AuthenticationProvider of type JwtAuthenticationProvider.
3. JwtAuthenticationProvider decodes, verifies, and validates JWT using a JwtDecoder.
4. JwtAuthenticationProvider then uses the JwtAuthenticationConverter to convert the Jwt into a Collection of granted authorities.
5. When authentication is successful, the authentication returned is of type JwtAuthenticationToken and has a principal that is the Jwt returned by the configured JwtDecoder. Ultimately, the returned JwtAuthenticationToken is set on SecurityContextHolder by the authentication filter.

The following are some of the advantages of JWT.

- **Compact**: JWTs are compact and can be sent as URL parameters, in an HTTP header, or in cookies.
- **Self-contained**: The token contains all the necessary information, reducing the need to query a database for user information.
- **Decentralized**: Since JWTs are self-contained, the server doesn't need to keep session information, making it easier to scale and distribute applications.

As per security considerations, JWTs are digitally signed, not encrypted. The information they contain can be decoded by anyone with access to the token, but the signature ensures its integrity. Storing sensitive data in the payload is not recommended, as the payload can be easily decoded.

To prevent tampering, it's important to use strong and secure algorithms for signing the tokens.

Secrets used for signing should be kept secret. If using public-key cryptography, the private key must be kept secure.

JWTs are widely used for building secure authentication and authorization mechanisms in modern web applications, APIs, and single sign-on (SSO) systems.

Here's how JWT and Spring Security can be integrated.

- **Dependency setup**: First, you must add the necessary dependencies to your project. Your project's build configuration should include Spring Security and libraries related to JWT.
- **Authentication and authorization configuration**: Configure Spring Security to manage authentication and authorization. This involves setting up security rules, authentication providers, and user details services. You can define which paths require authentication and which roles are required to access certain resources.
- **Token generation and validation**: Implement logic to generate and validate JWTs. Spring Security provides filters and classes to handle token-based authentication. You must create a mechanism to generate JWTs upon successful authentication and validate incoming JWTs for authorized requests.
- **Token processing filters**: Use Spring Security filters to intercept requests and perform authentication and authorization checks. You might implement a filter that examines the incoming JWT, validates it, and sets up the security context if the token is valid.
- **User details and authorities**: Extract user details and authorities from the JWT payload upon successful token validation. These details can be used to populate the Spring Security authentication context, allowing you to control access based on user roles and permissions.
- **Customizing authentication providers**: Depending on your authentication requirements, you might need to implement custom authentication providers to validate JWTs against your backend services or user databases.
- **Access control configuration**: Define access control rules using Spring Security's configuration. You can specify which roles are required to access different endpoints or resources. Roles and authorities can be derived from the JWT payload.
- **Exception handling**: Implement exception handling for cases where JWT validation fails, or unauthorized access is attempted. Customize error responses based on the security context.
- **Logout and token expiration**: For improved security, handle token expiration and implement a logout mechanism. JWTs can have an expiration time specified in their payload.
- **Testing and documentation**: Thoroughly test your JWT-based Spring Security implementation. Also, provide clear documentation for developers who work with the security system.

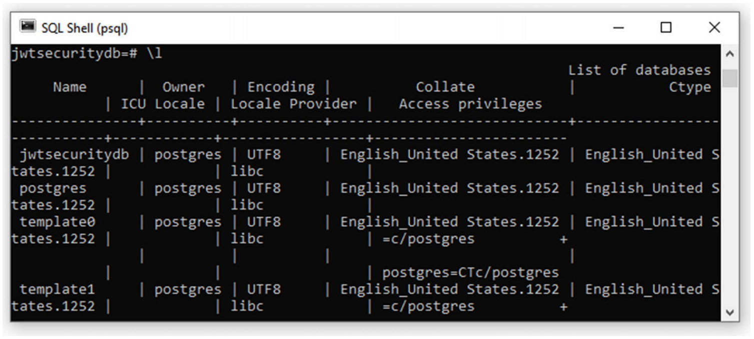
Keeping the Spring Security and JWT libraries updated is important because their APIs and best practices evolve. Spring Security's official documentation and online resources often provide detailed guides on integrating JWT-based authentication and authorization.

This chapter features an example showing how to secure a REST API using JWT using Spring Security 6, Spring Boot 3+, and PostgreSQL.

First, download and install PostgreSQL from
https://www.postgresql.org/download/windows/.

Next, create a new database named “jwtsecuritydb”. The username is “postgres” and the password is “postgres”.

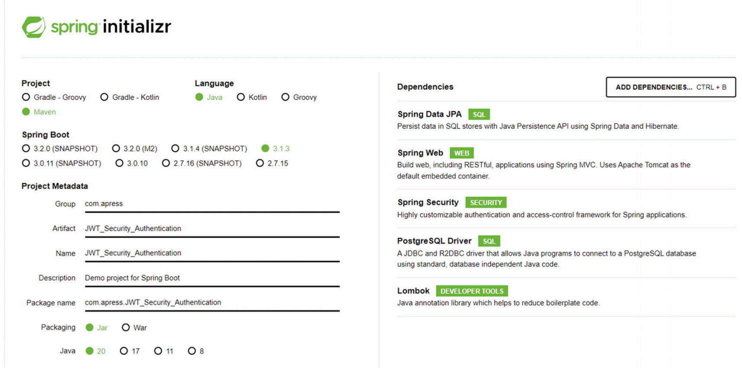
Figure **9-2** shows that our new PostgreSQL database is up and running.



```
jwtsecuritydb=# \l
                                                                  List of databases
     Name      |  Owner   | Encoding |          Collate           |            Ctype
      | ICU Locale | Locale Provider |   Access privileges
---------------+----------+----------+----------------------------+-----------------
----------+------------+-----------------+-----------------------
 jwtsecuritydb | postgres | UTF8     | English_United States.1252 | English_United S
tates.1252 |            | libc            |
 postgres      | postgres | UTF8     | English_United States.1252 | English_United S
tates.1252 |            | libc            |
 template0     | postgres | UTF8     | English_United States.1252 | English_United S
tates.1252 |            | libc            | =c/postgres          +
               |          |          |                            |
           |            |                 | postgres=CTc/postgres
 template1     | postgres | UTF8     | English_United States.1252 | English_United S
tates.1252 |            | libc            | =c/postgres          +
```

Figure 9-2 PostgreSQL shell console

Next, create a new Spring project named JWT_Security_Authentication using the Spring Initializr web tool at **https://start.spring.io/**, as shown in Figure **9-3**.



Project
Gradle - Groovy   Gradle - Kotlin   [✓ Maven]

Language
[✓ Java]   Kotlin   Groovy

Spring Boot
3.2.0 (SNAPSHOT)   3.2.0 (M2)   3.1.4 (SNAPSHOT)   [✓ 3.1.3]
3.0.11 (SNAPSHOT)   3.0.10   2.7.16 (SNAPSHOT)   2.7.15

Project Metadata

Group	com.apress
Artifact	JWT_Security_Authentication
Name	JWT_Security_Authentication
Description	Demo project for Spring Boot
Package name	com.apress.JWT_Security_Authentication
Packaging	[✓ Jar] War
Java	[✓ 20] 17 11 8

Dependencies

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

PostgreSQL Driver SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Figure 9-3 New Spring project using Spring Initializr

This example uses Java 20, Maven, and JAR, with Spring Web, PostgreSQL Driver, Spring Security, Spring Data JPA, and Lombok as dependencies.

The project’s file structure is shown in Figure **9-4**.

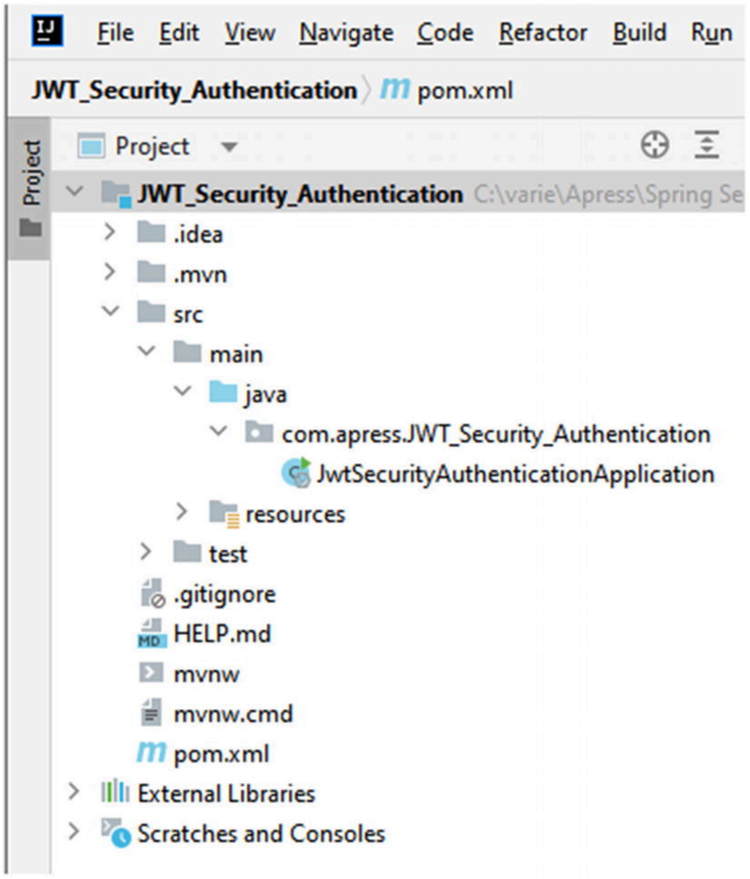

File Edit View Navigate Code Refactor Build Run
JWT_Security_Authentication pom.xml
Project
JWT_Security_Authentication C:\varie\Apress\Spring Se
.idea
.mvn
src
main
java
com.apress.JWT_Security_Authentication
JwtSecurityAuthenticationApplication
resources
test
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles

Figure 9-4 New Spring project structure

Next, add the needed dependencies in pom.xml files, such as JSON Web Token's io.jsonwebtoken and Jakarta XML Binding's jaxb-api. The entire pom.xml file is shown in Listing 9-1.

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"

    <modelVersion>4.0.0</modelVersion>

    <parent>

       <groupId>org.springframework.boot</groupId>

       <artifactId>spring-boot-starter-parent</artifactId>

       <version>3.1.3</version>
```

```
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.apress</groupId>
    <artifactId>JWT_Security_Authentication</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>JWT_Security_Authentication</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>20</java.version>
    </properties>
    <dependencies>
        <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-web</artifactId>

   </dependency>

    <dependency>

        <groupId>io.jsonwebtoken</groupId>

        <artifactId>jjwt</artifactId>

        <version>0.9.1</version>

    </dependency>       <dependency>

      <groupId>javax.xml.bind</groupId>

      <artifactId>jaxb-api</artifactId>

      <version>2.3.1</version>

   </dependency>




   <dependency>

      <groupId>org.postgresql</groupId>

      <artifactId>postgresql</artifactId>

      <scope>runtime</scope>

   </dependency>

   <dependency>

      <groupId>org.projectlombok</groupId>
```

```
            <artifactId>lombok</artifactId>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-test</artifactId>

            <scope>test</scope>

        </dependency>

        <dependency>

            <groupId>org.springframework.security</groupId>

            <artifactId>spring-security-test</artifactId>

            <scope>test</scope>

        </dependency>

    </dependencies>




    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Listing 9-1
The pom.xml File

Next, configure the application.properties file with information about the database used, the JPA/JWT, and the server configuration, as shown in Listing 9-2.

```
## DB Configuration ##
spring.datasource.url= jdbc:postgresql://localhost:5432/jwtsecuritydb
spring.datasource.username= postgres

spring.datasource.password= postgres



## JPA / HIBERNATE  Configuration ##

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

spring.jpa.generate-ddl=true



## Server Configuration ##

server.servlet.context-path=/api

server.port=8080



## JWT Configuration ##

jwt.jwtsecret = 2b44b0b00fd822d8ce753e54dac3dc4e06c2725f7db930f3b9924468b53194dbccdbe23d7baa5ef5fbc414c

jwt.jwtExpirationTime = 36000000



## User credentials

# to create user with "USER" Role: http://localhost:8080/api/user/register
```

```
#{
#"firstName": "Massimo",
#"lastName": "Nardone",
#"email": "mmassimo@gmail.com",
#"password": "masspasswd",
#"userRole": "user"
#}


# to login a user: http://localhost:8080/api/user/authenticate
#{
#"email": "mmassimo@gmail.com",
#"password": "masspasswd",
#}


## Admnin credentials
# to create user with "ADMIN" Role: http://localhost:8080/api/user/register
# {
# "firstName": "Neve",
# "lastName": "Nardon",
# "email": "neve@gmail.com",

# "password": "nevepasswd",

# "userRole": "admin"

# }



# to login as admin: http://localhost:8080/api/admin/hello

# {

# "email": "neve@gmail.com",

# "password": "nevepasswd"

# }
```

Listing 9-2 The application.properties File

This Spring Boot JWT authentication example registers a new user and logs in with a username and password. The user's role can be "admin" or "user" to authorize the user to access a certain resource.

The APIs included in our example are shown in Table 9-1.

Table 9-1 APIs Used in This Example

Method	URL	Action
POST	/api/user/register	Registers a new account
POST	/api/user/authenticate	Logs into an account
GET	/api/public/welcome	Retrieves public content
GET	/api/admin/hello	Accesses the admin's content

Let's create our user and role models.

First, define the roles and an enum called RoleName, as shown in Listing 9-3.

```
package com.apress.JWT_Security_Authentication.models;

public enum RoleName {

    USER, ADMIN;

}
```

Listing 9-3
The RoleName Class

Next, let’s define the role class, as shown in Listing **9-4**.

```
package com.ons.securitylayerJwt.models;

import jakarta.persistence.*;
import lombok.*;
import lombok.experimental.FieldDefaults;
import org.springframework.security.core.GrantedAuthority;
import java.io.Serializable;


@Entity

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

@FieldDefaults(level = AccessLevel.PRIVATE)

public class Role implements Serializable  {


    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    Integer id ;

    @Enumerated(EnumType.STRING)

    RoleName roleName ;


    public Role (RoleName roleName) {this.roleName = roleName;}

    public String getRoleName() {
        return roleName.toString();

    }

}
```

Listing 9-4
The Role Class

The role class creates a table named “Role” with two roles, “USER” and “ADMIN”, which define the credentials required to register a new user.

Finally, create the user model class, as shown in Listing **9-5**.

```
package com.apress.JWT_Security_Authentication.models;



import jakarta.persistence.*;

import lombok.*;

import lombok.experimental.FieldDefaults;

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.userdetails.UserDetails;



import java.io.Serializable;
import java.util.ArrayList;

import java.util.Collection;

import java.util.List;



@Entity



@Table(name = "users",

        uniqueConstraints = {

                @UniqueConstraint(columnNames = "firstName"),

                @UniqueConstraint(columnNames = "lastname"),

                @UniqueConstraint(columnNames = "email")

        })



@Getter

@Setter

@AllArgsConstructor

@ToString

@NoArgsConstructor

@FieldDefaults(level = AccessLevel.PRIVATE)

public class User implements Serializable , UserDetails {
```

```
@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

Integer id ;

String firstName ;

String lastName ;

String email;

String password ;

String userRole ;



@ManyToMany(fetch = FetchType.EAGER  , cascade = CascadeType.PERSIST)

List <Role> roles ;



public User (String email , String password , List<Role> roles) {

  this.email= email ;

  this.password=password ;

  this.roles=roles ;}



@Override
```

```
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> authorities = new ArrayList<>();
        this.roles.forEach(role -> authorities.add(new SimpleGrantedAuthority(role.getRoleName())));
        return authorities;
    }


    @Override
    public String getUsername() {
        return this.email;
    }


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


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

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

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

Listing 9-5
The User Model Class

The user class is mainly a model to fetch and validate all the user credentials if they are not expired, locked, or enabled.

Next, implement the repositories needed by each model for persisting and accessing data. In the repository package, let's create two repositories.

- UserRepository fetches the user repository information, as shown in Listing **9-6**.
- RoleRepository fetches the role repository information, as shown in Listing **9-7**.

```
package com.apress.JWT_Security_Authentication.repository;

import com.apress.JWT_Security_Authentication.models.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User,Integer> {

    Boolean existsByEmail(String email);
    Optional<User> findByEmail(String email);

}
```

Listing 9-6
The UserRepository Class

```
package com.apress.JWT_Security_Authentication.repository;

import com.apress.JWT_Security_Authentication.models.Role;
import com.apress.JWT_Security_Authentication.models.RoleName;
import org.springframework.data.jpa.repository.JpaRepository;

public interface RoleRepository extends JpaRepository<Role,Integer> {

    Role findByRoleName(RoleName roleName);

}
```

Listing 9-7
The RoleRepository Class

Let’s configure the SpringSecurityConfig class to use the security package, as shown in Listing **9-8**.

```
package com.apress.JWT_Security_Authentication.security;
```

```
import lombok.RequiredArgsConstructor;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfig

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

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

@EnableWebSecurity

@RequiredArgsConstructor

public class SpringSecurityConfig {



    private final JwtAuthenticationFilter jwtAuthenticationFilter ;

    private final CustomerUserDetailsService customerUserDetailsService ;
```

```
@Bean

public SecurityFilterChain filterChain (HttpSecurity http) throws Exception

{ http

        .csrf().disable()

        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS

        .authorizeHttpRequests(auth ->

        auth.requestMatchers("/public/**", "/user/**").permitAll()

        .requestMatchers("/admin/**").hasAuthority("ADMIN")) ;



    http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);



    return  http.build();

}



@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfig

{ return authenticationConfiguration.getAuthenticationManager();}
```

```
    @Bean

    public PasswordEncoder passwordEncoder()

    { return new BCryptPasswordEncoder(); }




}
```

Listing 9-8
The SpringSecurityConfig Class

Since this is the most important Spring Security class, let's discuss it in more detail.

- **@EnableWebSecurity**: allows Spring to find and automatically apply the class to the global web security.
- Spring Security loads the user details to perform authentication and authorization. It has `customerUserDetailsService` interface that you need to implement.
- **PasswordEncoder** used for the AuthenticationProvider if specified, it uses plain text.
- **(HttpSecurity http)** method used from WebSecurityConfigurerAdapter interface to tell Spring Security how to configure CSRF (disabled to send POST API), which filter (`jwtAuthenticationFilter`) and when you want it to work (filter before UsernamePasswordAuthenticationFilter), which Exception Handler is chosen (`JwtUtilities`).
- The implementation of `customerUserDetailsService` configures AuthenticationProvider with the **AuthenticationManagerBuilder.userDetailsService()** method.
- The "/public/**" path as a simple GET API is permitted to everyone so that you can test a simple GET API with public content.
- The "/users/**" path as a POST API is permitted to everyone so that all users can register and log in.
- The "/admin/**" path as a GET API is permitted only to users with hasAuthority("ADMIN"))

Listing 9-9 shows the CustomerUserDetailsService class.

```
package com.apress.JWT_Security_Authentication.security;

import com.apress.JWT_Security_Authentication.models.User;
import com.apress.JWT_Security_Authentication.repository.UserRepository;
import lombok.RequiredArgsConstructor;
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.Component;

@Component
@RequiredArgsConstructor
public class CustomerUserDetailsService implements UserDetailsService {

    private final UserRepository UserRepository ;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = UserRepository.findByEmail(email).orElseThrow(()-> new UsernameNotFoundException("U

        return  user ;

    }

}
```

Listing 9-9
The CustomerUserDetailsService Class

Now let’s create the JWT authentication filter and the authentication provider to make the security filter chain work.

The `JwtAuthenticationFilter` class is a filter that executes once per request (see Listing **9-10**).

```
package com.apress.JWT_Security_Authentication.security;

import jakarta.servlet.FilterChain;

import jakarta.servlet.ServletException;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

import org.springframework.lang.NonNull;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;




import java.io.IOException;




@Slf4j

@Component

@RequiredArgsConstructor

public class JwtAuthenticationFilter extends OncePerRequestFilter {




   private  final JwtUtilities jwtUtilities ;

   private final CustomerUserDetailsService customerUserDetailsService ;




    @Override
    protected void doFilterInternal(@NonNull HttpServletRequest request,

                                    @NonNull HttpServletResponse response,

                                    @NonNull FilterChain filterChain)

                                    throws ServletException, IOException {




        String token = jwtUtilities.getToken(request) ;




        if (token!=null && jwtUtilities.validateToken(token))

        {

            String email = jwtUtilities.extractUsername(token);




            UserDetails userDetails = customerUserDetailsService.loadUserByUsername(email);

            if (userDetails != null) {

            UsernamePasswordAuthenticationToken authentication =

                    new UsernamePasswordAuthenticationToken(userDetails.getUsername() ,null , userDetai

                log.info("authenticated user with email :{}", email);

            SecurityContextHolder.getContext().setAuthentication(authentication);




        }
        }

            filterChain.doFilter(request,response);

    }

}
```

Listing 9-10
The JwtAuthenticationFilter Class

Let's go over the `JwtAuthenticationFilter` class.

Create the JWT service class used in the `JwtAuthenticationFilter` class.

First, make sure the authorization header from our request is not null and that it starts with the word *bearer*.

Next, if the request has JWT, validate it and parse the username from it. Extract our JWT from the authorization header and use a function from the JwtSecvice class called extractUsername to extract the value of the user email from the JWT.

Next, from the username, use UserDetails to create an Authentication object and set the current UserDetails in SecurityContext using the setAuthentication(authentication) method.

Finally, send to get UserDetails.

```
UserDetails userDetails = customerUserDetailsService.loadUserByUsername(email);
```

Let's create the JwtUtilities class in the .security.jwt package, where you do the following.

- extract username from JWT: `extractUsername(String token)`

- generate a JWT from email, date, expiration, secret
- validate a JWT: invalid signature, expired JWT token, unsupported JWT token, and so on

Listing 9-11 shows the JwtUtilities class.

```
package com.apress.JWT_Security_Authentication.security;



import io.jsonwebtoken.*;

import jakarta.servlet.http.HttpServletRequest;

import lombok.extern.slf4j.Slf4j;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Component;

import org.springframework.util.StringUtils;



import java.time.Instant;

import java.time.temporal.ChronoUnit;

import java.util.Date;

import java.util.List;

import java.util.function.Function;
```

```
@Slf4j

@Component

public class JwtUtilities{



    @Value("${jwt.jwtsecret}")

    private String jwtsecret;



    @Value("${jwt.jwtExpirationTime}")

    private Long jwtExpirationTime;



    public String extractUsername(String token) {

        return extractClaim(token, Claims::getSubject);

    }



    public Claims extractAllClaims(String token) {return Jwts.parser().setSigningKey(jwtsecret).parseCl



    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {

        final Claims claims = extractAllClaims(token);

        return claimsResolver.apply(claims);
    }

    public Date extractExpiration(String token) { return extractClaim(token, Claims::getExpiration); }



    public Boolean validateToken(String token, UserDetails userDetails) {

        final String email = extractUsername(token);

        return (email.equals(userDetails.getUsername()) && !isTokenExpired(token));

    }

    public Boolean isTokenExpired(String token) {

        return extractExpiration(token).before(new Date());

    }



    public String generateToken(String email , List<String> roles) {



        return Jwts.builder().setSubject(email).claim("role",roles).setIssuedAt(new Date(System.current

                .setExpiration(Date.from(Instant.now().plus(jwtExpirationTime, ChronoUnit.MILLIS)))

                .signWith(SignatureAlgorithm.HS256, jwtsecret).compact();

    }



    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(jwtsecret).parseClaimsJws(token);
            return true;
        } catch (SignatureException e) {
            log.info("Invalid JWT signature.");
            log.trace("Invalid JWT signature trace: {}", e);
        } catch (MalformedJwtException e) {
            log.info("Invalid JWT token.");
            log.trace("Invalid JWT token trace: {}", e);
        } catch (ExpiredJwtException e) {
            log.info("Expired JWT token.");
            log.trace("Expired JWT token trace: {}", e);
        } catch (UnsupportedJwtException e) {
            log.info("Unsupported JWT token.");
            log.trace("Unsupported JWT token trace: {}", e);
        } catch (IllegalArgumentException e) {
            log.info("JWT token compact of handler are invalid.");
            log.trace("JWT token compact of handler are invalid trace: {}", e);
        }
        return false;
```

```
    }



    public String getToken (HttpServletRequest httpServletRequest) {

         final String bearerToken = httpServletRequest.getHeader("Authorization");

         if(StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer "))

         {return bearerToken.substring(7,bearerToken.length()); } // The part after "Bearer "

         return null;

    }



}
```

Listing 9-11
The JwtUtilities Class

Let’s create the TDO classes, including the following.

- BearerToken sets the JWT bearer token used in our example (see Listing 9-12).
- LoginDto is the data transfer object for user login (see Listing 9-13).
- RegisterDto is the data transfer object for user registration (see Listing 9-14).

```
package com.ons.securitylayerJwt.dto;
```

```
import lombok.Data;

@Data
public class BearerToken {

    private String accessToken ;
    private String tokenType ;

    public BearerToken(String accessToken , String tokenType) {
        this.tokenType = tokenType ;
        this.accessToken = accessToken;
    }

}
```

Listing 9-12
The BearerToken Class

```
package com.ons.securitylayerJwt.dto;

import lombok.AccessLevel;

import lombok.Data;

import lombok.experimental.FieldDefaults;

@Data

@FieldDefaults(level = AccessLevel.PRIVATE)

public class LoginDto {

    private String email ;

    private String password ;

}
```

Listing 9-13
The LoginDto Class

```
package com.ons.securitylayerJwt.dto;
```

```
import lombok.AccessLevel;

import lombok.Data;

import lombok.experimental.FieldDefaults;



import java.io.Serializable;



@Data

@FieldDefaults(level = AccessLevel.PRIVATE)

public class RegisterDto implements Serializable {



    String firstName ;

    String lastName ;

    String email;

    String password ;

    String userRole ;

}
```

Listing 9-14
The RegisterDto Class

Let’s create the Spring REST API Controller classes, including the following.

- PublicRestController is a simple REST GET API with a “/public/welcome” link to return a welcome message (see Listing 9-15).
- AdminRestController is a REST GET API with the “/admin/hello” link to return a Welcome admin message in case the email/password are the admin’s correct credential, the user has an “ADMIN” role, and the valid JWT is provided (see Listing 9-16).
- UserRestController is two REST POST APIs to register and log in a user. This is discussed more in Listing 9-17.

```
package com.apress.JWT_Security_Authentication.presentation;



import lombok.RequiredArgsConstructor;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;



@RestController

@RequestMapping("/public")

@RequiredArgsConstructor

public class PublicRestController {



    @GetMapping("/welcome")

    public String welcome ()
    { return "Welcome! This is a public content!" ;}

}
```

Listing 9-15
The PublicRestController Class

```
package com.apress.JWT_Security_Authentication.presentation;

import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/admin")
@RequiredArgsConstructor
public class AdminRestController {
    @GetMapping("/hello")

    public String sayHello ()

    { return "Welcome you are authenticated as Admin!" ;}



}
```

Listing 9-16
The AdminRestController Class

```
package com.apress.JWT_Security_Authentication.presentation;



import com.apress.JWT_Security_Authentication.controllers.IUserService;

import com.apress.JWT_Security_Authentication.dto.LoginDto;

import com.apress.JWT_Security_Authentication.dto.RegisterDto;



import lombok.RequiredArgsConstructor;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;



@RestController

@RequestMapping("/user")

@RequiredArgsConstructor

public class UserRestController {



    private final IUserService iUserService ;



    @PostMapping("/register")

    public ResponseEntity<?> register (@RequestBody RegisterDto registerDto)

    { return  iUserService.register(registerDto);}



    @PostMapping("/authenticate")

    public String authenticate(@RequestBody LoginDto loginDto)

    { return  iUserService.authenticate(loginDto);}
}
```

Listing 9-17
The UserRestController Class

The REST API `UserService` controller and its `IUserService` interface register a new user in the database and log in the user.

Finally, let's create the Spring REST APIs controller (see Listings 9-18 and 9-19) for authentication, providing APIs for register and login actions, such as the following.

- api/user/register
  - checks existing username/email
  - creates a new user (with "USER" or "ADMIN" role based on register input)
  - saves User to database using UserRepository
- api/user/authenticate
  - authenticates email, password
  - updates SecurityContext using the Authentication object
  - generates JWT
  - gets UserDetails from the Authentication object
  - response contains JWT and UserDetails data

```
package com.apress.JWT_Security_Authentication.controllers;



import com.apress.JWT_Security_Authentication.dto.LoginDto;

import com.apress.JWT_Security_Authentication.dto.RegisterDto;

import com.apress.JWT_Security_Authentication.models.User;

import com.apress.JWT_Security_Authentication.models.Role;

import org.springframework.http.ResponseEntity;
```

```
public interface IUserService {



   String authenticate(LoginDto loginDto);

   ResponseEntity<?> register (RegisterDto registerDto);

   Role saveRole(Role role);



   User saverUser (User user) ;

}
```

Listing 9-18
The IUserService Class

```
package com.apress.JWT_Security_Authentication.controllers;



import com.apress.JWT_Security_Authentication.dto.LoginDto;

import com.apress.JWT_Security_Authentication.dto.RegisterDto;

import com.apress.JWT_Security_Authentication.dto.BearerToken;
import com.apress.JWT_Security_Authentication.models.User;

import com.apress.JWT_Security_Authentication.models.Role;

import com.apress.JWT_Security_Authentication.models.RoleName;

import com.apress.JWT_Security_Authentication.repository.RoleRepository;

import com.apress.JWT_Security_Authentication.repository.UserRepository;

import com.apress.JWT_Security_Authentication.security.JwtUtilities;

import jakarta.transaction.Transactional;

import lombok.RequiredArgsConstructor;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Service;



import java.util.ArrayList;

import java.util.Collections;
import java.util.List;


@Service

@Transactional

@RequiredArgsConstructor

public class UserService implements IUserService{


    private final AuthenticationManager authenticationManager ;

    private final UserRepository userRepository ;

    private final RoleRepository roleRepository ;

    private final PasswordEncoder passwordEncoder ;

    private final JwtUtilities jwtUtilities ;


    @Override

    public Role saveRole(Role role) {

        return roleRepository.save(role);

    }


    @Override
    public User saverUser(User user) {

        return userRepository.save(user);

    }




    @Override

    public ResponseEntity<?> register(RegisterDto registerDto) {

        if(userRepository.existsByEmail(registerDto.getEmail()))

        { return  new ResponseEntity<>("email is already taken !", HttpStatus.SEE_OTHER); }

        else

        { User user = new User();

            user.setEmail(registerDto.getEmail());

            user.setFirstName(registerDto.getFirstName());

            user.setLastName(registerDto.getLastName());

            user.setPassword(passwordEncoder.encode(registerDto.getPassword()));

            String myrole = "user";




            if (registerDto.getUserRole().equals("") || registerDto.getUserRole().equals("user")) {

                myrole = "USER";

            }
            if (registerDto.getUserRole().equals("admin")) {

                myrole = "ADMIN";

            }




            Role role = roleRepository.findByRoleName(RoleName.valueOf(myrole));




            user.setUserRole(registerDto.getUserRole());




            user.setRoles(Collections.singletonList(role));

            userRepository.save(user);

            String token = jwtUtilities.generateToken(registerDto.getEmail(),Collections.singletonList(

            return new ResponseEntity<>(new BearerToken(token , "Bearer "),HttpStatus.OK);



        }

        }



    @Override

    public String authenticate(LoginDto loginDto) {
      Authentication authentication= authenticationManager.authenticate(

                new UsernamePasswordAuthenticationToken(

                        loginDto.getEmail(),

                        loginDto.getPassword()

                )

        );

        SecurityContextHolder.getContext().setAuthentication(authentication);

        User user = userRepository.findByEmail(authentication.getName()).orElseThrow(() -> new Username

        List<String> rolesNames = new ArrayList<>();

        user.getRoles().forEach(r-> rolesNames.add(r.getRoleName()));

        String token = jwtUtilities.generateToken(user.getUsername(),rolesNames);




        return "User login successful! Token: " + token;




    }




}
```

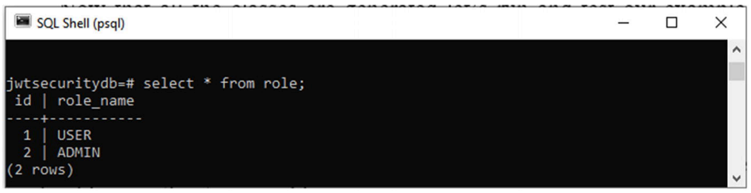
Listing 9-19
The IUserService Class

The last Java class to update is `JwtSecurityAuthenticationApplication`, shown in Listing 9-20, where the user and admin roles are populated automatically into the roles database table, as shown in Figure 9-5.

```
package com.apress.JWT_Security_Authentication;



import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.security.crypto.password.PasswordEncoder;


import com.apress.JWT_Security_Authentication.controllers.IUserService;

import com.apress.JWT_Security_Authentication.models.Role;

import com.apress.JWT_Security_Authentication.models.RoleName;

import com.apress.JWT_Security_Authentication.repository.RoleRepository;

import com.apress.JWT_Security_Authentication.repository.UserRepository;


@SpringBootApplication

public class JwtSecurityAuthenticationApplication {
    public static void main(String[] args) {

       SpringApplication.run(JwtSecurityAuthenticationApplication.class, args);

    }



    @Bean

    CommandLineRunner run (IUserService iUserService , RoleRepository roleRepository , UserRepository u

    {return  args ->

    {   iUserService.saveRole(new Role(RoleName.USER));

       iUserService.saveRole(new Role(RoleName.ADMIN));



    };}



}
```

Listing 9-20

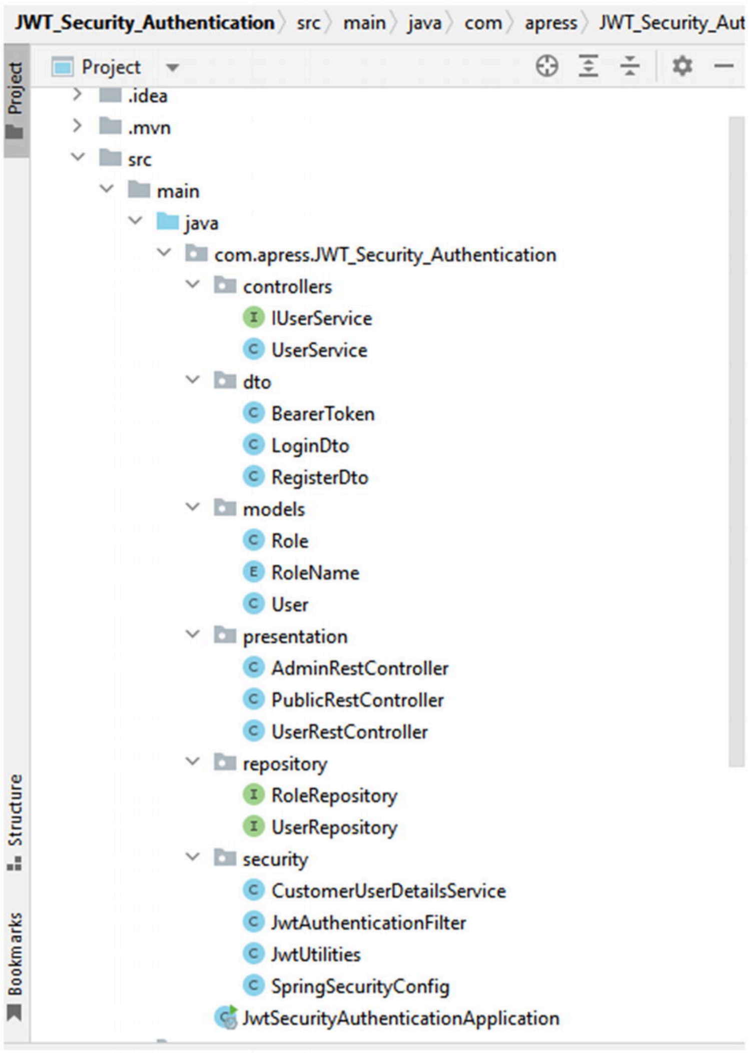
The JwtSecurityAuthenticationApplication Class



```
jwtsecuritydb=# select * from role;
 id | role_name
----+-----------
  1 | USER
  2 | ADMIN
(2 rows)
```

Figure 9-5 Spring user roles added into database

The new project structure should look like Figure 9-6.

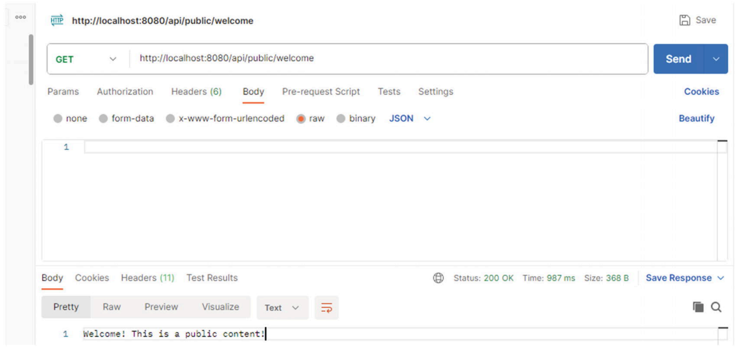


```
JWT_Security_Authentication > src > main > java > com > apress > JWT_Security_Aut
Project
  > .idea
  > .mvn
  v src
    v main
      v java
        v com.apress.JWT_Security_Authentication
          v controllers
              IUserService
              UserService
          v dto
              BearerToken
              LoginDto
              RegisterDto
          v models
              Role
              RoleName
              User
          v presentation
              AdminRestController
              PublicRestController
              UserRestController
          v repository
              RoleRepository
              UserRepository
          v security
              CustomerUserDetailsService
              JwtAuthenticationFilter
              JwtUtilities
              SpringSecurityConfig
            JwtSecurityAuthenticationApplication
```

Figure 9-6 Final Spring project structure

Now that all the classes are generated, let's run and test our example: `mvn spring-boot:run`

Let's test the http://localhost:8080/api/public/welcome to see that the public REST GET API works properly. Figure 9-7 shows the result using the Postman testing tool.


http://localhost:8080/api/public/welcome
Save
GET
http://localhost:8080/api/public/welcome
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary JSON
Beautify
Body Cookies Headers (11) Test Results
Status: 200 OK Time: 987 ms Size: 368 B
Save Response
Pretty Raw Preview Visualize Text

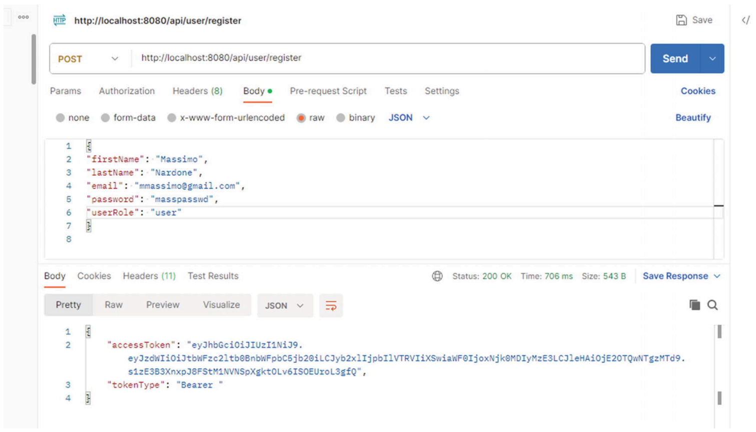
```
Welcome! This is a public content!
```

Figure 9-7 Testing public REST GET API

Next, let's register, via http://localhost:8080/api/user/register, a new user with the "USER" role and the following credentials.

```
{
"firstName": "Massimo",
"lastName": "Nardone",
"email": "mmassimo@gmail.com",
"password": "masspasswd",
"userRole": "user"
}
```

Figure 9-8 shows the result in Postman.

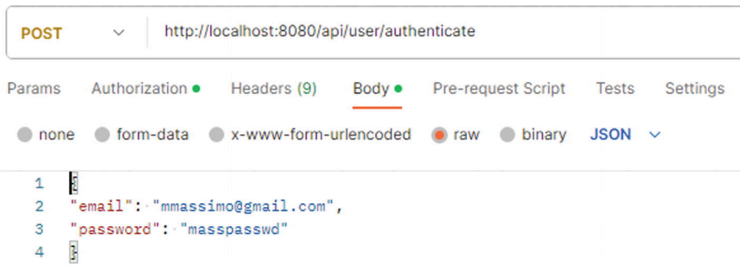

http://localhost:8080/api/user/register
Save
POST
http://localhost:8080/api/user/register
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary JSON
Beautify
Body Cookies Headers (11) Test Results
Status: 200 OK Time: 706 ms Size: 543 B
Save Response
Pretty Raw Preview Visualize JSON

```
{
"firstName": "Massimo",
"lastName": "Nardone",
"email": "mmassimo@gmail.com",
"password": "masspasswd",
"userRole": "user"
}
```

```
{
    "accessToken": "eyJhbGciOiJIUzI1NiJ9.
        eyJzdWIiOiJtbWFzc2ltb0BnbWFpbC5jb20iLCJyb2xlIjpbIlVTRVIiXSwiaWF0IjoxNjk0MDIyMzE3LCJleHAiOjE2OTQwNTgzMTd9.
        s1zE3B3XnxpJ8FStM1NVNSpXgktOLv6ISOEUroL3gfQ",
    "tokenType": "Bearer "
}
```

Figure 9-8 Registered new user with "user" role

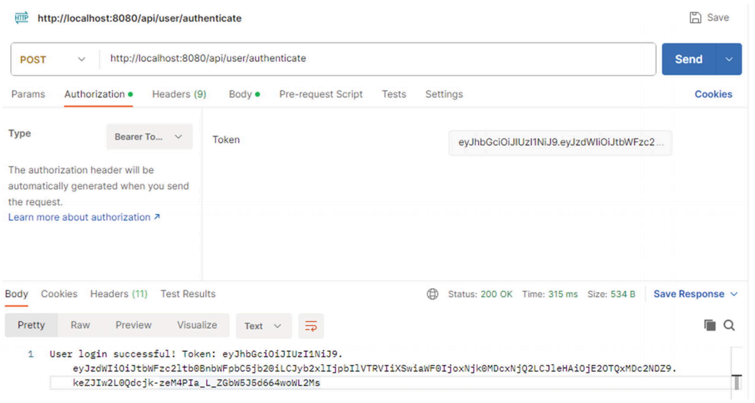
A new user is registered with status 200 OK, and an access JSON web token is generated. The token type is Bearer. You can use that token to log in, providing an email/password and the newly created JSON token, via http://localhost:8080/api/user/authenticate. Figures 9-9 and 9-10 show the results.



http://localhost:8080/api/user/authenticate

```
{
"email": "mmassimo@gmail.com",
"password": "masspasswd"
}
```

Figure 9-9 Log in user with valid email/password



http://localhost:8080/api/user/authenticate

POST http://localhost:8080/api/user/authenticate

Type: Bearer To...

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

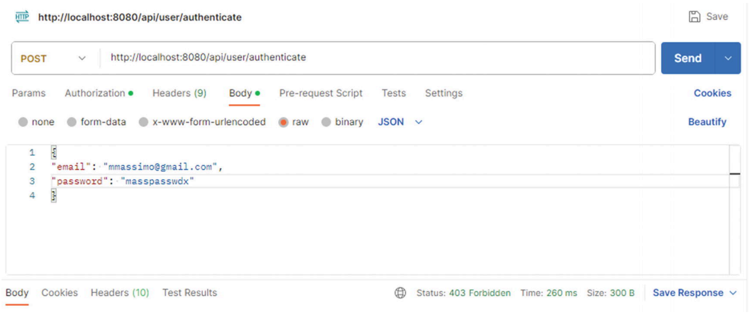
Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJtbWFzc2...

Status: 200 OK Time: 315 ms Size: 534 B

```
User login successful! Token: eyJhbGciOiJIUzI1NiJ9.
    eyJzdWIiOiJtbWFzc2ltb0BnbWFpbC5jb20iLCJyb2xlIjpbIlVTRVIiXSwiaWF0IjoxNjk0MDcxNjQ2LCJleHAiOjE2OTQxMDc2NDZ9.
    keZJIw2L0Qdcjk-zeM4PIa_L_ZGbW5J5d664woWL2Ms
```

Figure 9-10 Log in user with valid JWT

Status 200 OK validates the user login. Figure 9-11 shows the result when a wrong password is entered.



http://localhost:8080/api/user/authenticate

POST http://localhost:8080/api/user/authenticate

```
{
"email": "mmassimo@gmail.com",
"password": "masspasswdx"
}
```

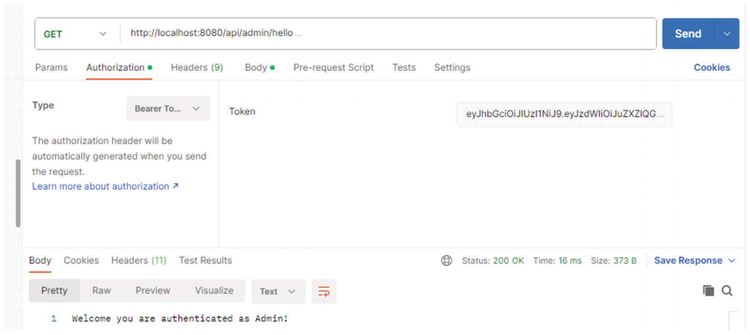
Status: 403 Forbidden Time: 260 ms Size: 300 B

Figure 9-11 Forbidden login for user providing wrong password

Let’s create another user with the “ADMIN” role.

```
{

"firstName": "Neve",
"lastName": "Nardon",

"email": "neve@gmail.com",

"password": "nevepasswd",

"userRole": "admin"

}
```

If you try to log in using http://localhost:8080/api/admin/hello to the user API using the admin credential, it fails because we defined that only users with an admin role can access the admin URL. If instead you log in providing valid email/password credentials and a valid JWT, you see what is shown in Figure 9-12.



GET http://localhost:8080/api/admin/hello ...

Type: Bearer To...

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

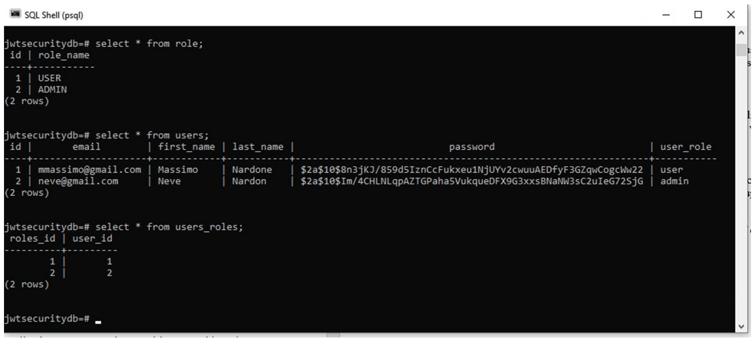
Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWliOiJuZXZIQG...

Status: 200 OK Time: 16 ms Size: 373 B

```
Welcome you are authenticated as Admin!
```

Figure 9-12 Log in to Admin REST API

The users and roles created in the database are shown in Figure 9-13.



```
jwtsecuritydb=# select * from role;
 id | role_name
----+-----------
  1 | USER
  2 | ADMIN
(2 rows)


jwtsecuritydb=# select * from users;
 id |        email        | first_name | last_name |                           password                           | user_role
----+---------------------+------------+-----------+--------------------------------------------------------------+-----------
  1 | mmassimo@gmail.com | Massimo    | Nardone   | $2a$10$8n3jKJ/859d5IznCcFukxeu1NjUYv2cwuuAEDfyF3GZqwCogcWw22 | user
  2 | neve@gmail.com     | Neve       | Nardon    | $2a$10$Im/4CHLNLqpAZTGPaha5VukqueDFX9G3xxsBNaNW3sC2uIeG72SjG | admin
(2 rows)


jwtsecuritydb=# select * from users_roles;
 roles_id | user_id
----------+---------
        1 |       1
        2 |       2
(2 rows)


jwtsecuritydb=# _
```

Figure 9-13 Created users in the database

Summary

This chapter showed you how to secure a REST API using JWT and Spring Security 6, Spring Boot 3+, and PostgreSQL.