

# Implementing Spring Security 6 with Spring Boot 3: A Guide to OAuth and JWT with Nimbus for Authentication

**DEV** [dev.to/pryhmez/implementing-spring-security-6-with-spring-boot-3-a-guide-to-oauth-and-jwt-with-nimbus-for-authentication-2lhf](https://dev.to/pryhmez/implementing-spring-security-6-with-spring-boot-3-a-guide-to-oauth-and-jwt-with-nimbus-for-authentication-2lhf)

## Introduction

Since the introduction of Spring Security 6, I have encountered many developers who experienced issues getting their heads around setting it up to serve their business needs.

Therefore if

- You are new to spring security
- or you have been using older versions of Spring Security and Spring Boot and are finding it difficult to implement Spring Security on Spring Boot 3 with Spring Security 6.
- You are looking for a simpler way to set up spring security so that you don't have to install external libraries for JWT and create complete filters.

Then this article is for you.

First, let's dive into the basics of spring security and what is required to set up spring security using Nimbus for JWT.

## What is spring security

According to the definition in springs documentation Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de facto standard for securing Spring-based applications.

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

some of the features of spring security include

- Comprehensive and extensible support for both Authentication and Authorization
- Protection against attacks like session fixation, clickjacking, cross-site request forgery, etc
- Servlet API integration
- Optional integration with Spring Web MVC
- Much more...

it is important to use Spring security because it includes updated security features hence ensuring your application has up-to-date security features.

## Project setup

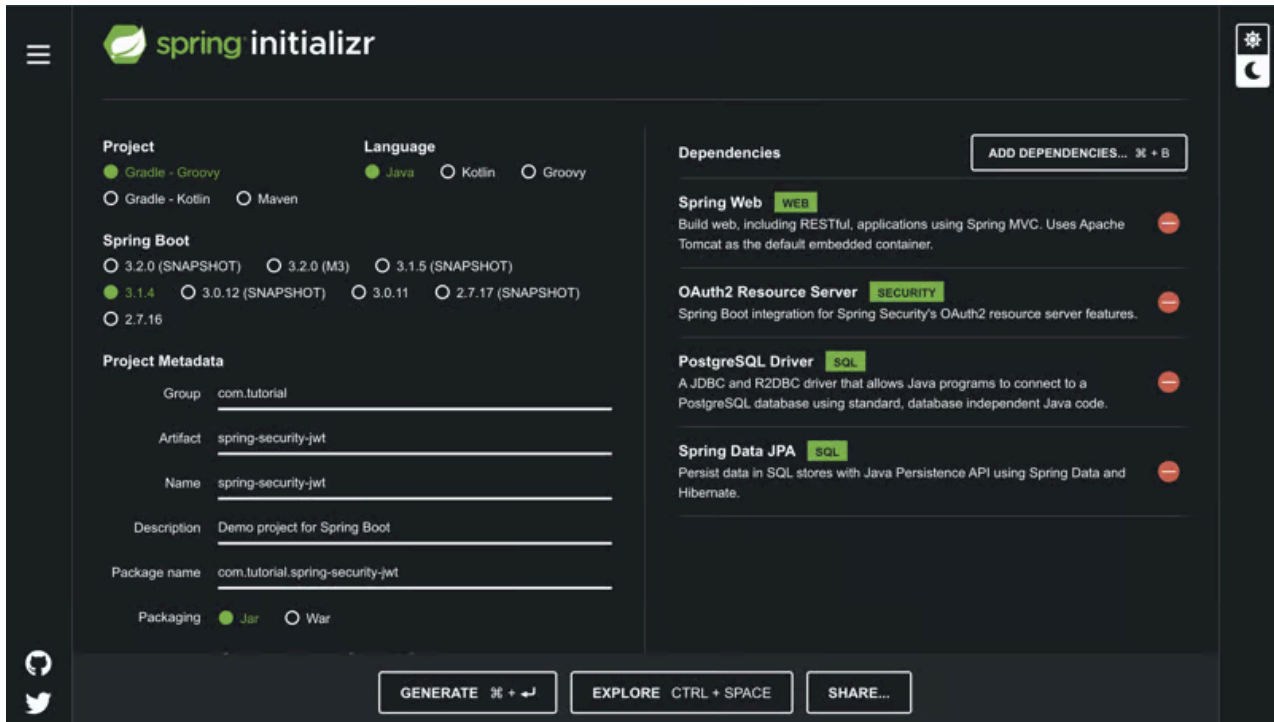
Alright, enough with all the theory with no action. Let's go straight into setting up our project.

Now if you have stayed even a short while with spring you should know every spring developer's special website spring-initializer

start.spring.io

for initializing our spring projects. <https://start.spring.io/>

We are going to add the following dependencies

The screenshot shows the Spring Initializr web application interface. It has a dark theme. On the left, there's a sidebar with a hamburger menu icon and social media icons for GitHub and Twitter. The main content area is divided into several sections: 'Project' with radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions 3.2.0 (SNAPSHOT), 3.2.0 (M3), 3.1.5 (SNAPSHOT), 3.1.4 (selected), 3.0.12 (SNAPSHOT), 3.0.11, and 2.7.17 (SNAPSHOT), and 2.7.16; 'Project Metadata' with input fields for Group (com.tutorial), Artifact (spring-security-jwt), Name (spring-security-jwt), Description (Demo project for Spring Boot), and Package name (com.tutorial.spring-security-jwt), plus a 'Packaging' section with radio buttons for 'Jar' (selected) and 'War'; and a 'Dependencies' section on the right with a list of selected dependencies: 'Spring Web' (WEB), 'OAuth2 Resource Server' (SECURITY), 'PostgreSQL Driver' (SQL), and 'Spring Data JPA' (SQL). Each dependency has a red minus icon to its right. At the bottom, there are three buttons: 'GENERATE' (with a keyboard shortcut ⌘ + ↵), 'EXPLORE' (with a keyboard shortcut CTRL + SPACE), and 'SHARE...'. A settings icon is in the top right corner.

So we added

1. Spring web for building web APIs
2. OAuth2 Resource Server for security
3. Spring data JPA since we are going to be making use of storage to store user data
4. PostgreSQL Driver because we are going to be using the Postgres database.

**Note:** Notice how we did not use spring security instead we used OAuth2 Resource Server That is because i discovered that OAuth2 Resource Server contains Nimbus which could be used for generating and managing JWTs without the need for extra dependency additions. Also, note that I used spring boot 3.1.4.

Now once we are done with setting up our dependencies we download the jar file, extract our project, and open it in any IDE of our choice(in my case I chose IntelliJ).

## Setting up application config

---

Now we need to setup our application configuration and also setup a connection to our **Postgres** database(that's if you are using Postgres, but if you decide to use an in-memory database like H2 you can check online for how to connect although you will still have to do that in the application.yaml or application.properties based on which you prefer).

```
spring:
  application:
    name: spring-security-jwt
  datasource:
    url: jdbc:postgresql://localhost:5432/spring-security
    username:
    password:
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
        format_sql: true
    database: postgresql
    database-platform: org.hibernate.dialect.PostgreSQLDialect

server:
  port: 8000
  error:
    include-message: always

rsa:
  private-key: classpath:certs/private-key.pem
  public-key: classpath:certs/public-key.pem

logging:
  level:
    org:
      springframework: INFO
```

## Creating public and private keys for encryption and decryption

Now in the resources directory, we create a folder called certs and then open our terminal and navigate into that directory running this command

```
cd src/main/resources/certs
```

Then we will be using OpenSSL to generate an RSA-key keypair(this should come default for Mac users and can also be set for other users)

Generate a Private Key (RSA):

```
openssl genpkey -algorithm RSA -out private-key.pem
```

This command generates an RSA private key and saves it to the private-key.pem file.

Extract the Public Key from the Private Key by running:

```
openssl rsa -pubout -in private-key.pem -out public-key.pem
```

Then convert it to the appropriate PKCS format and replace the old one

```
openssl pkcs8 -topk8 -inform PEM -outform PEM -in private-key.pem -out private-key.pem -nocrypt
```

Alright, if that last step was all good let's move on to the next step.

Recall, that in the application.yaml file there was a section of configuration that looked like this.

```
rsa:
  private-key: classpath:certs/private-key.pem
  public-key: classpath:certs/public-key.pem
```

all we just did was tell spring where to find our public and private keys for encrypting and decrypting our JWT tokens.

## Setting up our User modules and the using the RSAKeys in our application

---

Now that we are done creating our RSAKeys using OpenSSL, what we have to do now is help spring-boot use it through configuration properties.

1. first we need to create a package called **config** then we create a file in the config package we created earlier and name it **RsaKeyConfigProperties** and then paste the code below inside it

```
package com.tutorial.springsecurityjwt.config;

import org.springframework.boot.context.properties.ConfigurationProperties;

import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

@ConfigurationProperties(prefix = "rsa")
public record RsaKeyConfigProperties(RSAPublicKey publicKey, RSAPrivateKey privateKey ) {
}
```

1. Now we will create our **user** module to manage all user activities then we will create the **User.java** class that will serve as our entity.

```
package com.tutorial.springsecurityjwt.user;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.micrometer.common.lang.NonNull;
import jakarta.persistence.*;

import java.util.*;

@Entity
@Table(name = "users", uniqueConstraints = {
    @UniqueConstraint(columnNames = "user_name"),
    @UniqueConstraint(columnNames = "email")
})
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(unique = true)
    private String userId;

    @Column(name = "user_name", unique = true)
    @NonNull
    private String username;

    @NonNull
    @Column(name = "email", unique = true)
    private String email;

    @NonNull
    @JsonIgnore
    private String password;

    public User() {
    }

    public User(String userId, @NonNull String username, @NonNull String email, @NonNull
String password) {
        this.userId = userId;
        this.username = username;
        this.email = email;
        this.password = password;
    }

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    @NonNull
    public String getUsername() {
        return username;
    }
```

```

    }

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

    @NonNull
    public String getEmail() {
        return email;
    }

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

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

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

    @Override
    public String toString() {
        return "User{" +
            "userId='" + userId + '\'' +
            ", username='" + username + '\'' +
            ", email='" + email + '\'' +
            ", password='" + password + '\'' +
            '}';
    }
}

```

```

}

```

1. We will then create our **UserRepository** interface to handle JPA database interaction and queries, it will extend JpaRepository. paste the following code below.

```

package com.tutorial.springsecurityjwt.user;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

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

```

## Setting up our Auth Modules

Let's setup our AuthUser which will be used to manage user permissions and roles.  
Paste the following code in the AuthUser.java class in the auth module

```
package com.tutorial.springsecurityjwt.auth;

import com.tutorial.springsecurityjwt.user.User;
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.stream.Collectors;

public class AuthUser extends User implements UserDetails {

    private final User user;

    public AuthUser(User user) {
        this.user = user;
    }

    public User getUser() {
        return user;
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

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

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

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

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

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

    @Override
    public boolean isEnabled() {
```



```
        return true;
    }
}
```

We will create our `AuthServices.java` class which will manage all logic that enable authentication. paste the following code below

```
package com.tutorial.springsecurityjwt.auth;

import com.tutorial.springsecurityjwt.user.UserRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.oauth2.jwt.JwtClaimsSet;
import org.springframework.security.oauth2.jwt.JwtEncoder;
import org.springframework.security.oauth2.jwt.JwtEncoderParameters;
import org.springframework.stereotype.Service;

import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.stream.Collectors;

@Service
public class AuthService {

    private static final Logger log = LoggerFactory.getLogger(AuthService.class);
    @Autowired
    private JwtEncoder jwtEncoder;
    @Autowired
    private PasswordEncoder passwordEncoder;
    @Autowired
    private UserRepository userRepository;

    public String generateToken(Authentication authentication) {
        Instant now = Instant.now();

        String scope = authentication.getAuthorities()
            .stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.joining(" "));

        JwtClaimsSet claims = JwtClaimsSet.builder()
            .issuer("self")
            .issuedAt(now)
            .expiresAt(now.plus(10, ChronoUnit.HOURS))
            .subject(authentication.getName())
            .claim("scope", scope)
            .build();

        return jwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
    }
}
```

We will also create a class in the auth package **JpaUserDetailsService.java** which will handle loading users from the database for login.

```
package com.tutorial.springsecurityjwt.auth;

import com.tutorial.springsecurityjwt.user.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;

@Service
public class JpaUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        AuthUser user = userRepository
            .findByUsername(username)
            .map(AuthUser::new)
            .orElseThrow(() -> new UsernameNotFoundException("User name not found: " +
username));

        return user;
    }
}
```

### ***Setting up our spring security config***

In our config package create a java class named **SecurityConfig.java** and paste the following code inside.

```
package com.tutorial.springsecurityjwt.config;

import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.source.ImmutableJWKSet;
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;
import com.tutorial.springsecurityjwt.auth.JpaUserDetailsService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.ProviderManager;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
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.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtEncoder;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;
import org.springframework.security.oauth2.jwt.NimbusJwtEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.servlet.handler.HandlerMappingIntrospector;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    private static final Logger log = LoggerFactory.getLogger(SecurityConfig.class);

    @Autowired
    private RsaKeyConfigProperties rsaKeyConfigProperties;

    @Autowired
    private JpaUserDetailsService userDetailsService;

    @Bean
    public AuthenticationManager authManager() {

        var authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return new ProviderManager(authProvider);
    }
}
```

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http, HandlerMappingIntrospector
introspector) throws Exception {

    return http
        .csrf(csrf -> {
            csrf.disable();
        })
        .cors(cors -> cors.disable())
        .authorizeHttpRequests(auth -> {
            auth.requestMatchers("/error/**").permitAll();
            auth.requestMatchers("/api/auth/**").permitAll();
            auth.anyRequest().authenticated();
        })
        .sessionManagement(s ->
s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .oauth2ResourceServer((oauth2) -> oauth2.jwt((jwt) ->
jwt.decoder(jwtDecoder()))
        .userDetailsService(userDetailsService)
        .httpBasic(Customizer.withDefaults())
        .build());
}

@Bean
public JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(rsaKeyConfigProperties.publicKey()).build();
}

@Bean
JwtEncoder jwtEncoder() {
    JWK jwk = new
RSAKey.Builder(rsaKeyConfigProperties.publicKey()).privateKey(rsaKeyConfigProperties.priva
teKey()).build();

    JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwks);
}

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

```

N.B: you have to add this line of code

`@EnableConfigurationProperties(RsaKeyConfigProperties.class)`

to your base application, in my case SpringSecurityJwtApplication.java. Else your application won't pick up your configuration.

Your base application should look something like this.

```
package com.tutorial.springsecurityjwt;

import com.tutorial.springsecurityjwt.config.RsaKeyConfigProperties;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.EnableConfigurationProperties;

@EnableConfigurationProperties(RsaKeyConfigProperties.class)
@SpringBootApplication
public class CollaboMainApplication {

    public static void main(String[] args) {
        SpringApplication.run(CollaboMainApplication.class, args);
    }
}
```

## Setting up the controller

---

Now we need to set up our auth rest controller to have routes for login and in a real-life case signup etc.

But for this example, we would just be setting up a login route and we would create a hard-coded user in the database.

In other to do this we have to first create our **AuthDTO**(DTO means data transfer object) so that we can receive login username and password then we set up our **AuthController.java** in our auth module/package and then paste the following code inside.

```
package com.tutorial.springsecurityjwt.auth;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
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.validation.annotation.Validated;
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("/api/auth")
@Validated
public class AuthController {

    private static final Logger log = LoggerFactory.getLogger(AuthController.class);

    @Autowired
    private AuthService authService;
    @Autowired
    private AuthenticationManager authenticationManager;

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody AuthDTO.LoginRequest userLogin) throws
    IllegalArgumentException {
        Authentication authentication =
            authenticationManager
                .authenticate(new UsernamePasswordAuthenticationToken(
                    userLogin.username(),
                    userLogin.password()));
        SecurityContextHolder.getContext().setAuthentication(authentication);

        AuthUser userDetails = (AuthUser) authentication.getPrincipal();

        log.info("Token requested for user :{}", authentication.getAuthorities());
        String token = authService.generateToken(authentication);

        AuthDTO.Response response = new AuthDTO.Response("User logged in successfully",
token);

        return ResponseEntity.ok(response);
    }
}
```

Now we are done with the basic steps for setting up spring security and JWT with Nimbus, now we need to test and we need dummy users in the database. We are going to do this by using the commandline runner to create users at the start of our application. Hence we will create a bean in our

**SpringSecurityJwtApplication.java** class which is the entry point into our application. you can update you entry class with the code below.

```
package com.tutorial.springsecurityjwt;

import com.tutorial.springsecurityjwt.config.RsaKeyConfigProperties;
import com.tutorial.springsecurityjwt.user.User;
import com.tutorial.springsecurityjwt.user.UserRepository;
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.bcrypt.BCryptPasswordEncoder;

@EnableConfigurationProperties(RsaKeyConfigProperties.class)
@SpringBootApplication
public class SpringSecurityJwtApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringSecurityJwtApplication.class, args);
    }

    @Bean
    public CommandLineRunner initializeUser(UserRepository userRepository,
BCryptPasswordEncoder passwordEncoder) {
        return args -> {

            User user = new User();
            user.setUsername("exampleuser");
            user.setEmail("example@gmail.com");
            user.setPassword(passwordEncoder.encode("examplepassword"));

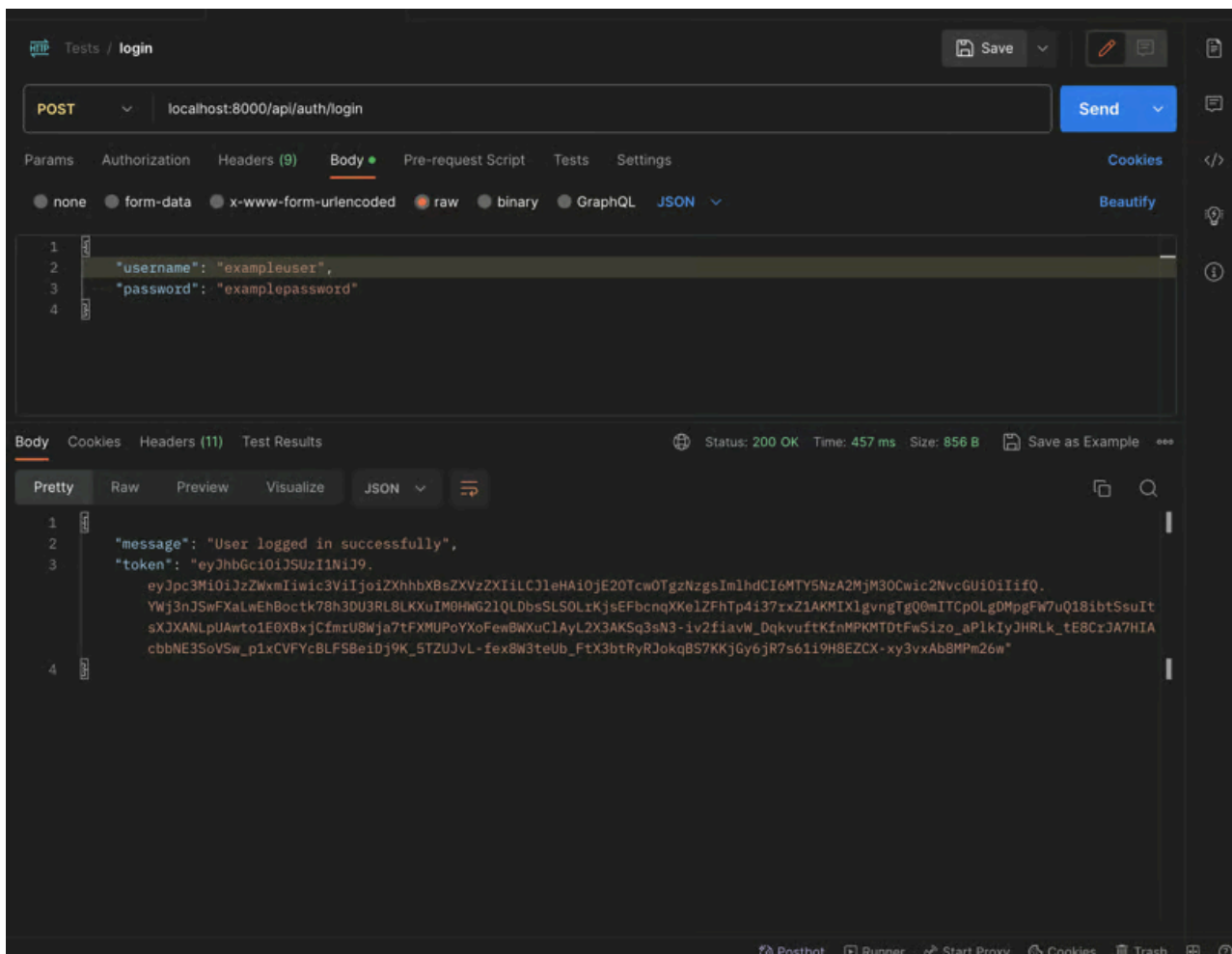
            // Save the user to the database
            userRepository.save(user);

        };
    }
}
```

## Testing our login and JWT

Now we are good to go all you have to do is to run your application and make sure you followed all the procedures and when your application starts without any errors you can test your login endpoint on Postman.





## Summary

A quick explanation of all the components of our setup and the authentication flow

1. We set up our project using springdataJPA, OAuth2 Resource Server, postgres driver, and spring web.
2. We created our user package, auth package, and config package.
3. We created public-keys and private-keys using OpenSSL for encrypting and decrypting JWTs and linked it to our application through the applicationConfig.yaml and setting up properties.
4. we created a user entity and a user repository for making JPA database calls.
5. we created an auth user that would manage roles, credentials, etc.
6. we created a JpaUserDetailsService for managing user details at sign-in, authService for managing auth logic like generating tokens.
7. We created a DTO to help us manage data transfers between the client and server request and response.
8. We used created an authController to manage routes for authentication requests like login.
9. Then we setup our Security configurations to used Nimbus to manage JWTs and also user our UserDetailsService to manage user details on sign-in.

Below is a picture of what the file structure should look like

