**Week-4**

**Spring REST**

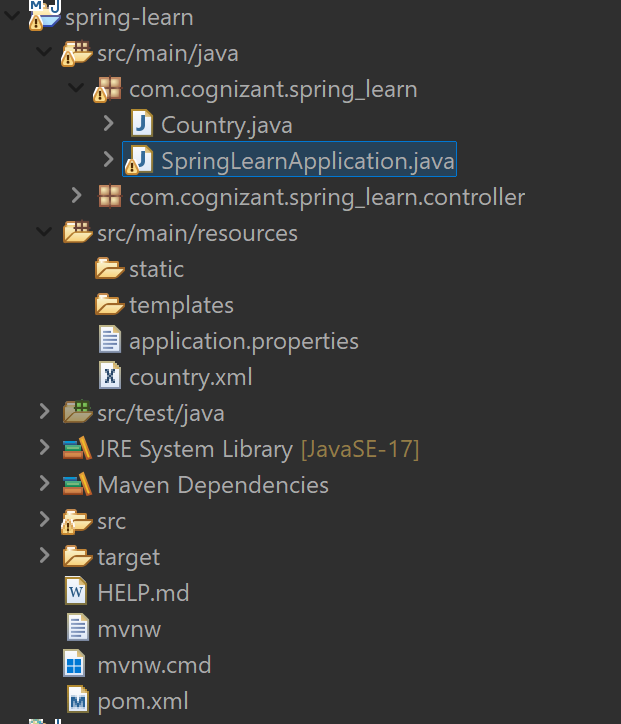
**Exercise 1: Create a Spring Web Project using Maven**

**Steps:**

1. **Set Up a Spring Project:**
   * Create a Maven project named spring\_learn.
   * Add Spring Core dependencies in the pom.xml file.
2. **Configure the Application Context:**
   * Create an XML configuration file named applicationContext.xml in the src/main/resources directory.
3. **Define SpringLearnApplication Classe:**
   * In the package com.spring\_learn add a class SpringLearnApplication.
4. **Run the Application:**
   * Create a main class to load the Spring context and test the configuration.

Spring Boot with Maven allows developers to quickly build and run stand-alone, production-ready Spring applications. Maven handles dependency management and project builds, while Spring Boot simplifies configuration with embedded servers and auto-configuration.

**Folder Structure:**

****

**SpringLearnApplication.java**

// SpringLearnApplication.java

package com.cognizant.spring\_learn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@SpringBootApplication // Important to enable component scan and Spring Boot features

public class SpringLearnApplication {

private static final Logger LOGGER = LoggerFactory.getLogger(SpringLearnApplication.class);

public static void main(String[] args) {

LOGGER.info("START");

//This starts the Spring Boot web application on the configured port (like 8081)

SpringApplication.run(SpringLearnApplication.class, args);

LOGGER.info("END");

    }

}

**pom.xml**

//pom.xml dependencies

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

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

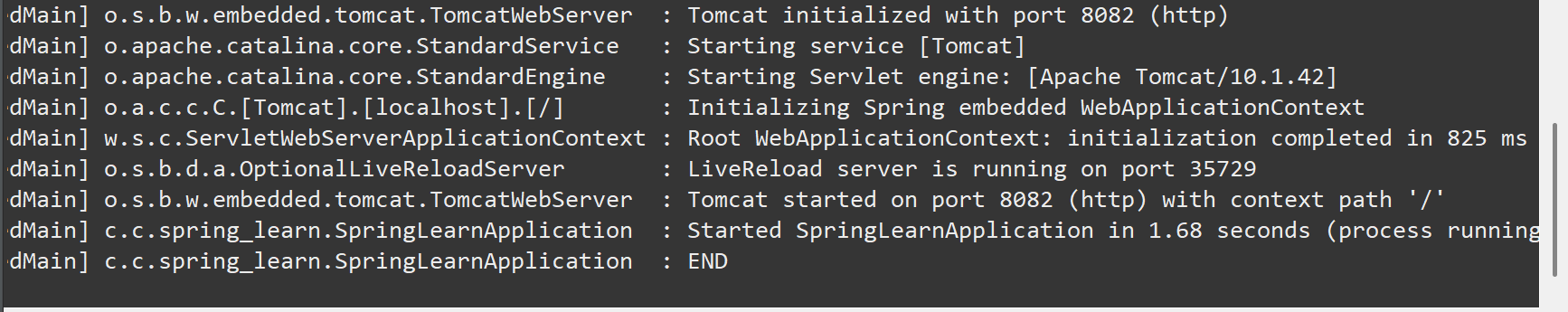
<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

**Output:**



**Exercise 2: Spring Core – Load Country from Spring Configuration XML**

**Steps:**

1. **Create the XML File:**
   * Create the country.xml file in the resources folder.
2. **Add the Country Class and modify the SpringLearnApplication:**
   * Ensure that Country class has the setter and getter methods.
3. **Test the Configuration:**
   * Run the SpringLearnApplication main class to verify the results in the console.

In Spring, XML configuration allows us to define beans and their properties externally, promoting loose coupling. Using ApplicationContext and ClassPathXmlApplicationContext, we can load these beans at runtime. This enables easy management and injection of objects without hardcoding them in Java code.

**Country.java**

// Country.java

public class Country {

private String code;

private String name;

public Country() {

System.out.println("Inside Country Constructor");

}

public void setCode(String code) {

System.out.println("Setting code: " + code);

this.code = code;

}

public void setName(String name) {

System.out.println("Setting name: " + name);

this.name = name;

}

public String toString() {

return "Country [code=" + code + ", name=" + name + "]";

}

}

**country.xml**

// country.xml

<bean id="country" class="com.cognizant.spring\_learn.Country">

<property name="code" value="IN" />

<property name="name" value="India" />

</bean>

**SpringLearnApplication.java**

// SpringLearnApplication.java

package com.cognizant.spring\_learn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

//package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringLearnApplication {

private static final Logger LOGGER = LoggerFactory.getLogger(SpringLearnApplication.class);

public static void main(String[] args) {

LOGGER.info("START");

displayCountry();

LOGGER.info("END");

}

public static void displayCountry() {

System.out.println("Before loading context");

ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");

System.out.println("After loading context");

Country country = context.getBean("country", Country.class);

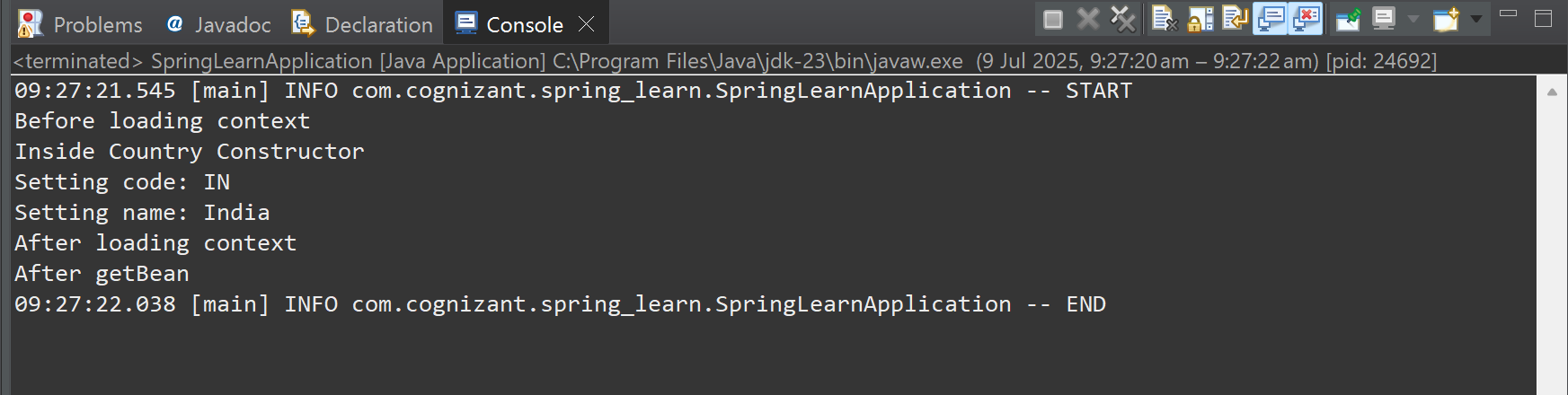
System.out.println("After getBean");

LOGGER.debug("Country : {}", country.toString());

}

}

**Output:**



**Exercise 3: Hello World RESTful Web Service**

**Steps:**

1. **Create a New Package:**
   * Create a new package as controller in the com.spring\_learn.
2. **Add a new Controller class:**
   * Include the new HelloController class in the package.
3. **Run the Application:**
   * Run the main class of the application and test the results.

RESTful Web Services are web services based on the REST (Representational State Transfer) architecture. They use standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources and return data in formats like JSON or XML. REST APIs are lightweight, scalable, and easy to integrate across platforms.

**HelloController.java**

//HelloController.java

package com.cognizant.spring\_learn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

@RestController

public class HelloController {

private static final Logger LOGGER = LoggerFactory.getLogger(HelloController.class);

@GetMapping("/hello")

public String sayHello() {

LOGGER.info("START: sayHello()");

String message = "Hello World!!";

LOGGER.info("END: sayHello()");

return message;

}

}

**Output:**



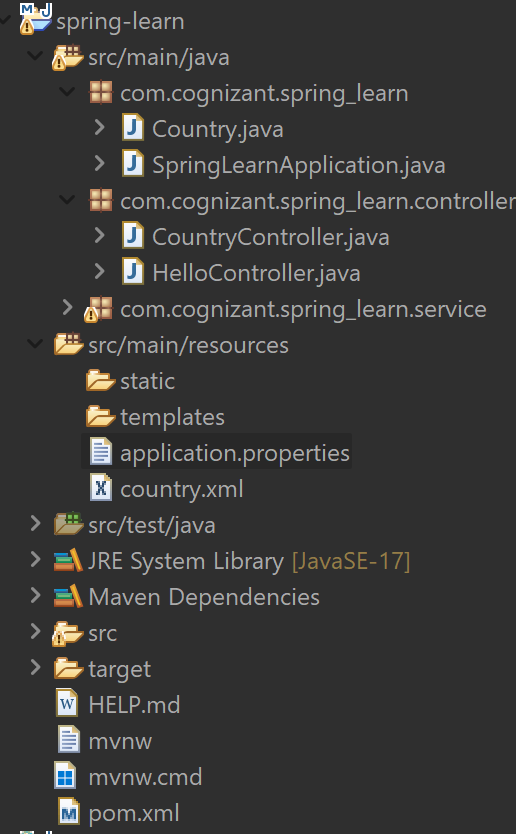
**Exercise 4: REST - Country Web Service**

**Steps:**

1. **Create a New Package:**
   * Create a new package as controller in the com.spring\_learn.
2. **Add a new Controller class:**
   * Include the new CountryController class in the package.
3. **Ensure we have the previous files.**
   * Make sure that we have the country.xml and country.java classes in the same project.
4. **Run the Application:**
   * Run the main class of the application and test the results.

Country-based Web Service is a RESTful service built using Spring Boot that returns country details like code and name. The country data is loaded from an XML configuration, and endpoints allow users to fetch either the default country (India) or a specific country using its code via path parameters.

**Folder Structure:**



**CountryController.java**

//CountryController.java

package com.cognizant.spring\_learn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

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

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

import com.cognizant.spring\_learn.model.Country;

@RestController

public class CountryController {

private static final Logger LOGGER = LoggerFactory.getLogger(CountryController.class);

@RequestMapping("/country")

public Country getCountryIndia() {

LOGGER.info("START: getCountryIndia");

ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");

Country country = context.getBean("country", Country.class);

LOGGER.info("END: getCountryIndia");

return country;

}

}

**Output:**



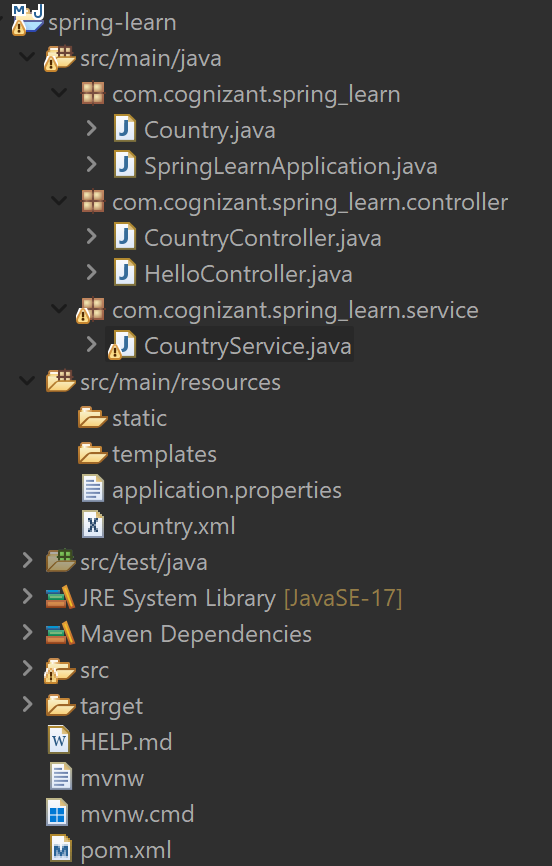
**Exercise 5: REST - Get country based on country code**

**Steps:**

1. **Create a New Package:**
   * Create a new package as service in the com.spring\_learn.
2. **Add a new Service class:**
   * Include the new CountyService class in the package.
3. **Ensure we have the previous files.**
   * Make sure that we have the country.xml and make appropriate changes to it as well as to CountryController.java.
4. **Run the Application:**
   * Run the main class of the application and test the results.

Get Country by Code Web Service is a RESTful API built using Spring Boot that retrieves a specific country’s details by its code. The controller accepts the country code as a path variable and calls a service method, which loads a list of countries from an XML file and returns the matching country using case-insensitive comparison.

**Folder Structure:**



**CountryController.java**

//CountryController.java

package com.cognizant.spring\_learn.controller;

import com.cognizant.spring\_learn.Country;

import com.cognizant.spring\_learn.service.CountryService;

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

import org.springframework.web.bind.annotation.\*;

@RestController

public class CountryController {

@Autowired

private CountryService countryService;

@GetMapping("/countries/{code}")

public Country getCountry(@PathVariable String code) {

return countryService.getCountry(code);

}

}

**CountryService.java**

//CountryService.java

package com.cognizant.spring\_learn.service;

import com.cognizant.spring\_learn.Country;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class CountryService {

public Country getCountry(String code) {

ApplicationContext context = new ClassPathXmlApplicationContext("country.xml");

List<Country> countryList = context.getBean("countryList", List.class);

return countryList.stream()

.filter(country -> country.getCode().equalsIgnoreCase(code))

.findFirst()

.orElse(null);

}

}

**country.xml**

//country.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="country" class="com.cognizant.spring\_learn.Country">

<property name="code" value="IN"/>

<property name="name" value="India"/>

</bean>

<bean id="countryList" class="java.util.ArrayList">

<constructor-arg>

<list>

<bean class="com.cognizant.spring\_learn.Country">

<property name="code" value="IN"/>

<property name="name" value="India"/>

</bean>

<bean class="com.cognizant.spring\_learn.Country">

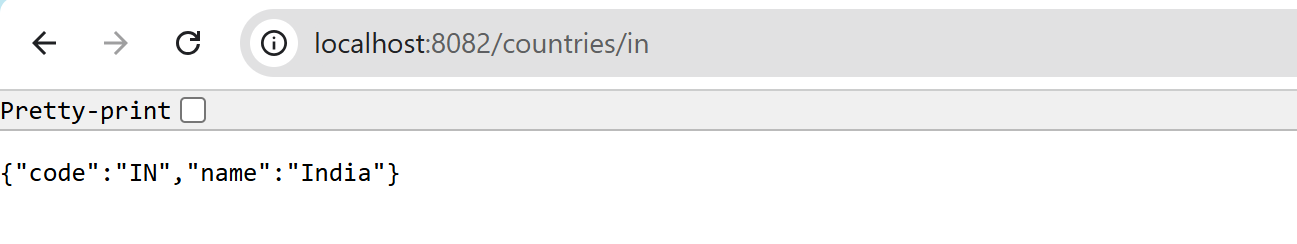
<property name="code" value="US"/>

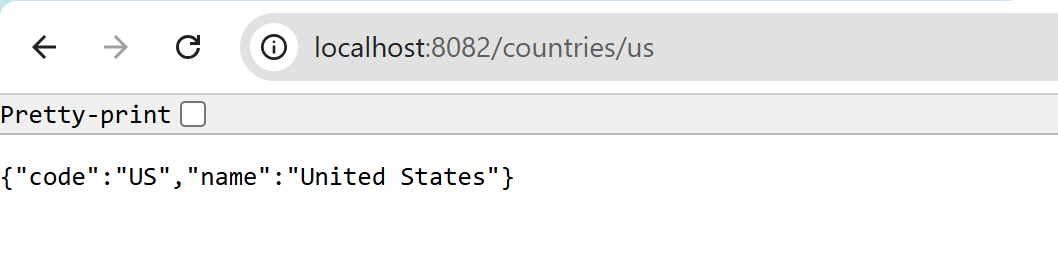
<property name="name" value="United States"/>

</bean>

</beans>

**Output:**





****

**Conclusion:**

In conclusion, through these Spring REST experiments, we gained hands-on experience in building robust RESTful APIs using Spring Boot. We explored key concepts such as controllers, service layers, dependency injection, and data exchange using JSON. These exercises provided a strong foundation for creating scalable, modular, and maintainable web services, which are essential in modern backend development.

**JWT** **(JSON Web Token)**

**Exercise 6: Create authentication service that returns JWT**

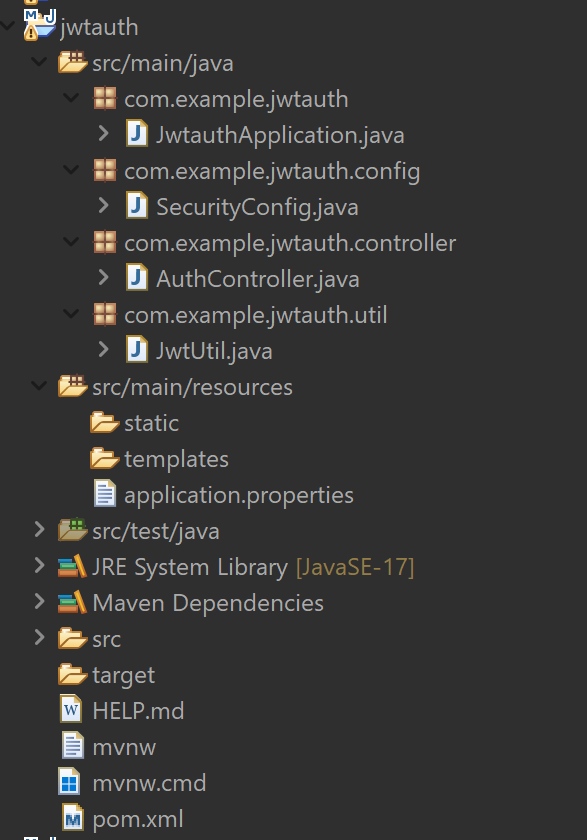
**Steps:**

1. **Setup Project & Dependencies:**
   * Create a Spring Boot Maven project and add dependencies: spring-boot-starter-web, spring-boot-starter-security, and jjwt.
2. **Configure Security:**
   * Create SecurityConfig.java to allow /authenticate endpoint and define in-memory user (user:pwd) for basic authentication.
3. **Create JWT Utility:**
   * Add JwtUtil.java to generate JWT tokens using a secret key, username, and expiration.
4. **Build Auth Controller:**
   * Create AuthController.java with /authenticate endpoint that reads credentials and returns a generated JWT.
5. **Run & Test:**
   * Run the main class of the application and test the results.

JWT (JSON Web Token) is a compact, URL-safe token used to securely transmit information between parties as a JSON object. It is signed using a secret or a public/private key pair to ensure data integrity and authenticity.

In authentication, JWT helps by issuing a token after verifying user credentials; this token is then sent with each request to access protected resources—eliminating the need to re-authenticate every time.

**Folder Structure:**



**AuthController.java**

//AuthController.java

package com.example.jwtauth.controller;

import com.example.jwtauth.util.JwtUtil;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import jakarta.servlet.http.HttpServletRequest;

import java.util.Base64;

@RestController

public class AuthController {

@Autowired

private JwtUtil jwtUtil;

//@GetMapping("/authenticate")

@GetMapping("/authenticate")

public ResponseEntity<?> authenticate(HttpServletRequest request) {

String authHeader = request.getHeader("Authorization");

if (authHeader == null || !authHeader.startsWith("Basic ")) {

return ResponseEntity.status(401).body("Missing Authorization header");

}

String base64Credentials = authHeader.substring("Basic ".length());

String credentials = new String(Base64.getDecoder().decode(base64Credentials));

String[] userDetails = credentials.split(":", 2);

String username = userDetails[0];

String password = userDetails[1];

if ("user".equals(username) && "pwd".equals(password)) {

String token = jwtUtil.generateToken(username);

return ResponseEntity.ok("{\"token\":\"" + token + "\"}");

} else {

return ResponseEntity.status(401).body("Invalid credentials");

}

}

}

**SecurityConfig.java**

//SecurityConfig.java

package com.example.jwtauth.config;

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.AuthenticationConfiguration;

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

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

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

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

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

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

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

import org.springframework.security.config.Customizer;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

return http

.csrf(csrf -> csrf.disable())

.authorizeHttpRequests(auth -> auth

.anyRequest().permitAll()

)

.httpBasic(Customizer.withDefaults())

.build();

}

@Bean

public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder) {

return new InMemoryUserDetailsManager(

User.withUsername("user")

.password(passwordEncoder.encode("pwd"))

.roles("USER")

.build()

);

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {

return config.getAuthenticationManager();

}

}

**JwtUtil.java**

//JwtUtil.java

package com.example.jwtauth.util;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import io.jsonwebtoken.security.Keys;

import org.springframework.stereotype.Component;

import java.security.Key;

import java.util.Date;

@Component

public class JwtUtil {

private static final Key SECRET\_KEY = Keys.hmacShaKeyFor("mysecretkeymysecretkeymysecretkey".getBytes());

public String generateToken(String username) {

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60)) // 1 hour

.signWith(SECRET\_KEY, SignatureAlgorithm.HS256)

.compact();

}

}

**JwtauthApplication.java**

//JwtauthApplication.java

package com.example.jwtauth;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // This must scan your config/controller/util packages

public class JwtauthApplication {

public static void main(String[] args) {

SpringApplication.run(JwtauthApplication.class, args);

}

}

**Pom.xml**

//pom.xml

<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>

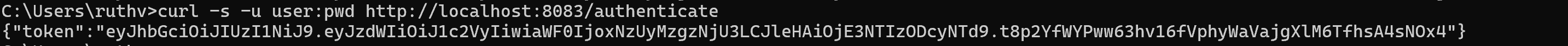
<artifactId>jjwt-jackson</artifactId> <!-- instead of jjwt-gson or jjwt-orgjson -->

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

**Output:**



**Conclusion:**

In conclusion, implementing JWT-based authentication in Spring Boot allows for stateless, secure, and scalable user authentication. By generating a signed token upon successful login, the server avoids maintaining session state, enhancing performance and simplicity. This approach is widely used in modern APIs and microservices where secure, token-based communication is essential.

**-- THE END --**