**WEEK 4 MANDATORY HANDS ON**

**Hands-on 1: Create a Spring Web Project using Maven**

***Objective***:

Create a basic Spring Boot web project using Maven and understand the structure and configurations.

***1. Project Initialization***

Go to https://start.spring.io/

Group: com.cognizant

Artifact: spring-learn

***Dependencies:***

Spring Boot DevTools

Spring Web

**Project Structure Overview**

***Path*** ***Description***

src/main/java Contains application source code.

src/main/resources Holds configuration files (e.g., XML, properties, YAML).

src/test/java Contains unit test cases.

**SpringLearnApplication.java Overview**

It is the entry point of the Spring Boot application.

Annotated with @SpringBootApplication which is a combination of:

* @Configuration: Marks as a configuration class.
* @EnableAutoConfiguration: Enables Spring Boot’s auto-configuration.
* @ComponentScan: Scans packages for components.

**pom.xml Walkthrough**

<groupId>com.cognizant</groupId>

<artifactId>spring-learn</artifactId>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

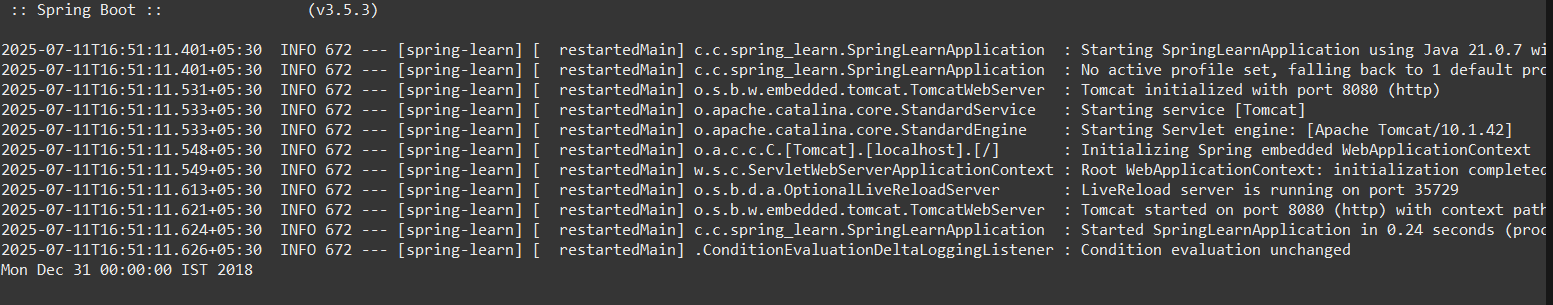
<groupId>org.springframework.boot</groupId>

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

<scope>runtime</scope>

</dependency>

</dependencies>



**Hands-on 4: Spring Core – Load Country from Spring Configuration XML**

**Objective**:

Load a bean from Spring XML configuration and demonstrate IoC (Inversion of Control).

**Steps to Implement**

***Country.java***

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Country {

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

private String code;

private String name;

public Country() {

LOGGER.debug("Inside Country Constructor.");

}

public String getCode() {

LOGGER.debug("Inside getCode");

return code;

}

public void setCode(String code) {

LOGGER.debug("Inside setCode");

this.code = code;

}

public String getName() {

LOGGER.debug("Inside getName");

return name;

}

public void setName(String name) {

LOGGER.debug("Inside setName");

this.name = name;

}

@Override

public String toString() {

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

}

}

**XML Configuration country.xml in src/main/resources**

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

<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

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

<bean id="country" class="com.cognizant.springlearn.Country">

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

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

</bean>

</beans>

***SpringLearnApplication.java***

package com.cognizant.springlearn;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

@SpringBootApplication

public class SpringLearnApplication {

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

public static void main(String[] args) {

LOGGER.info("START");

SpringApplication.run(SpringLearnApplication.class, args);

displayCountry();

LOGGER.info("END");

}

public static void displayCountry() {

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

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

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

}}

**Spring XML Configuration Tags**

**Tag Purpose**

<bean> Declares a bean managed by Spring container.

id Unique identifier for the bean.

class Fully qualified class name of the bean.

<property> Sets property values via setters.

name Name of the Java property.

value The value assigned to the property.

**ApplicationContext vs ClassPathXmlApplicationContext**

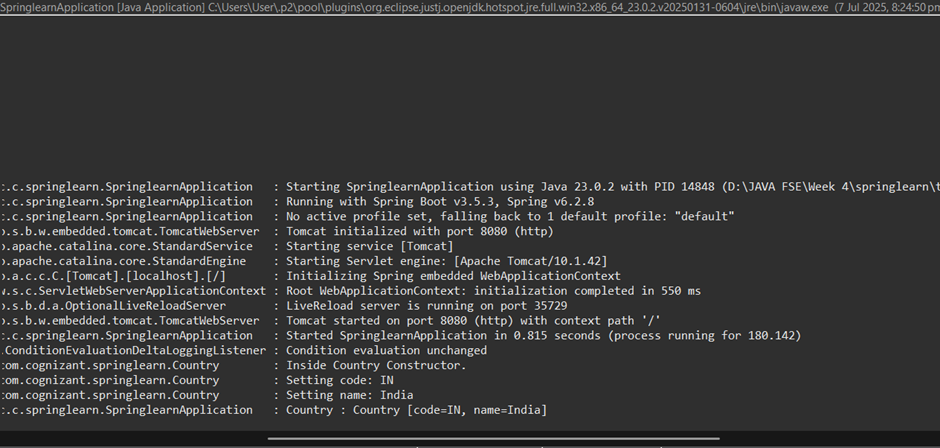
***Concept Explanation***

ApplicationContext Spring’s central interface for

accessing beans.

ClassPathXmlApplicationContext Concrete implementation to load:

beans from classpath XML files.



**Hello World RESTful Web Service**

URL: http://localhost:8083/hello

Method: GET

Response: "Hello World!!"

**Step-by-Step Implementation:**

1. Update application.properties in src/main/resources

properties

server.port=8083

[logging.level.com](http://logging.level.com).cognizant=DEBUG

**2. Create Controller: 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;

}

}

**Testing:**

In browser or Postman: http://localhost:8083/hello

**Response:**

Hello World!!

**REST - Return Single Country (India)**

Requirement:

URL: http://localhost:8083/country

Method: GET

Response: JSON of India from XML bean.

Step-by-Step Implementation:

**1. country.xml in src/main/resources**

<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

https://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>

</beans>

**2. CountryController.java**

@GetMapping("/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;

}

**Sample Response:**

{

"code": "IN",

"name": "India"

}

What Happens in Controller?

Spring invokes the getCountryIndia() method.

country.xml is parsed.

Bean with ID country is created and populated.

**REST - Return All Countries**

**Requirement:**

URL: http://localhost:8083/countries

Response: JSON array of all 4 countries.

**Step-by-Step Implementation:**

<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>

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

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

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

</bean>

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

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

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

</bean>

</list>

</constructor-arg>

</bean>

Controller Method:

@GetMapping("/countries")

public List<Country> getAllCountries() {

LOGGER.info("START - getAllCountries()");

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

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

LOGGER.info("END - getAllCountries()");

return countries;

}

Sample Response:

[

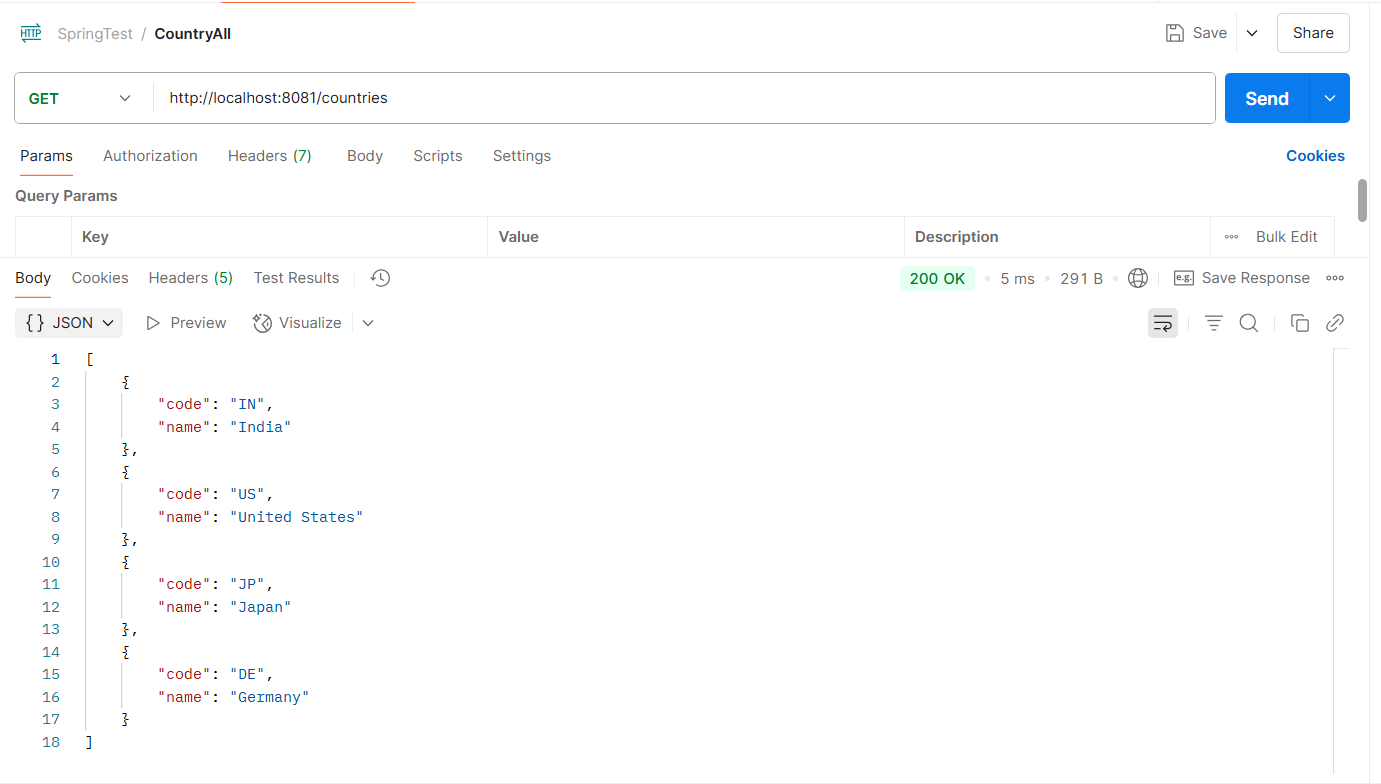
{ "code": "IN", "name": "India" },

{ "code": "US", "name": "United States" },

{ "code": "JP", "name": "Japan" },

{ "code": "DE", "name": "Germany" }

]



**REST - Get Country by Code**

Requirement:

URL: http://localhost:8083/countries/in

Response: JSON object matching country code

**Step-by-Step Implementation:**

**1. Create CountryService.java**

package com.cognizant.spring\_learn.service;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.stereotype.Service;

import com.cognizant.spring\_learn.Country;

@Service

public class CountryService {

public Country getCountry(String code) {

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

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

return countryList.stream()

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

.findFirst()

.orElse(null);

}

}

**2. Inject and Use in Controller:**

@Autowired

private CountryService countryService;

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

public Country getCountry(@PathVariable String code) {

LOGGER.info("START - getCountry({})", code);

Country country = countryService.getCountry(code);

LOGGER.info("END - getCountry()");

return country;

}

Sample Request:

http://localhost:8083/countries/in

Sample Response:

{

"code": "IN",

"name": "India"

}

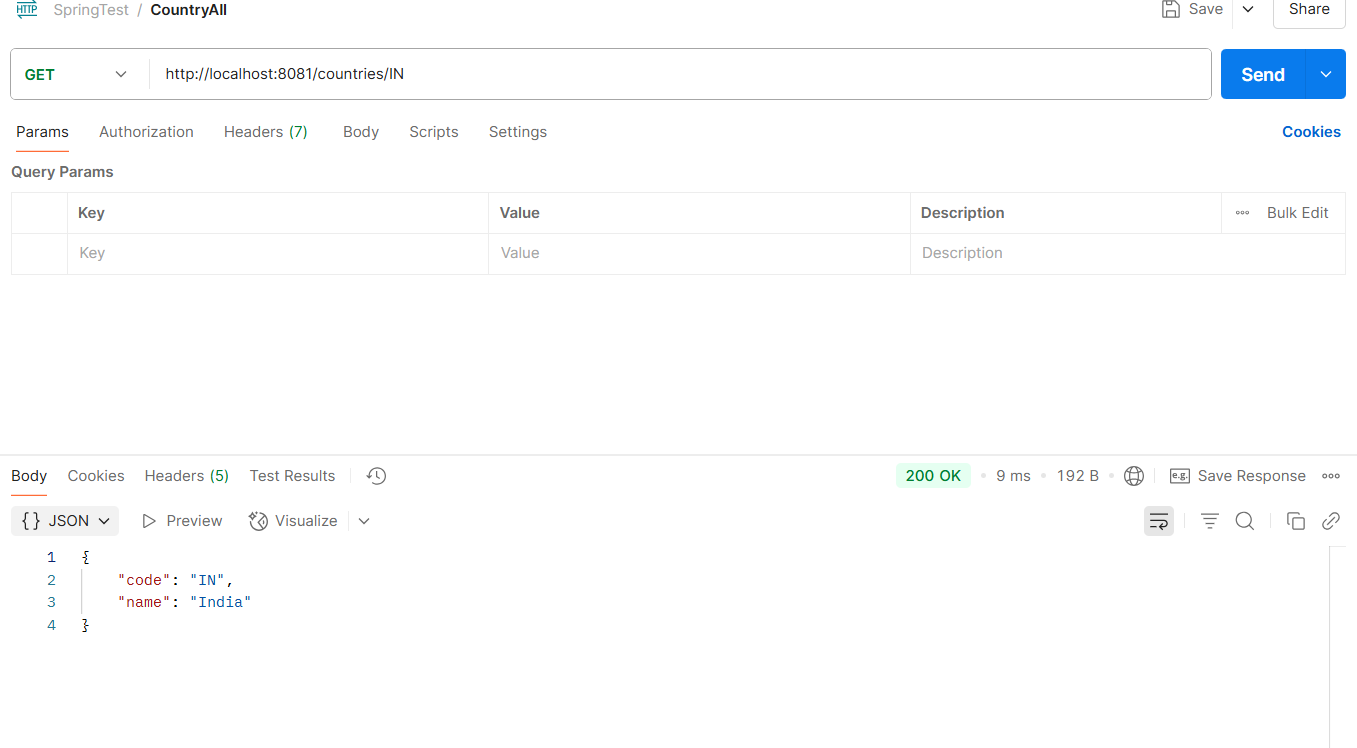
Request Mapping:

@GetMapping("/path") automatically maps GET requests.

@PathVariable extracts variable parts of the URI.

Behavior on Not Found:

If country is not found, it returns null with HTTP 200. You may enhance it later to throw a custom exception and return 404.



**JWT Authentication Service**

**Objective**

To create an **authentication service** that validates user credentials and returns a **JWT token** on successful authentication.

**Endpoint Details**

* **URL:** http://localhost:8090/authenticate
* **HTTP Method:** GET
* **Authentication Method:** Basic Auth (credentials via -u in curl)

**Example Request**

http://localhost:8090/authenticate

**Example Response**

{

"token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiaWF0IjoxNjcwMzc5NDc0LCJleHAiOjE2NzAzODA2NzR9.t3LRvlCV-hwKfoqZYlaVQqEUiBloWcWn0ft3tgv0dL0"

}

**Implementation Steps**

**Step 1️: Create Authentication Controller**

**Class:** com.cognizant.springlearn.controller.AuthenticationController

package com.cognizant.springlearn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

import java.util.Base64;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

@RestController

public class AuthenticationController {

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

private static final String SECRET\_KEY = "secretkey"; // Choose a stronger key in real scenarios

@GetMapping("/authenticate")

public Map<String, String> authenticate(@RequestHeader("Authorization") String authHeader) {

LOGGER.debug("START: authenticate()");

// Extract and decode credentials

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];

LOGGER.debug("Decoded username: {}", username);

// Here you can add logic to validate username and password from DB

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

String token = Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 10)) // 10 minutes

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

Map<String, String> tokenMap = new HashMap<>();

tokenMap.put("token", token);

LOGGER.debug("Generated Token: {}", token);

LOGGER.debug("END: authenticate()");

return tokenMap;

} else {

throw new RuntimeException("Invalid Credentials");

}

}

}

**Step 2️: Configure Security (SecurityConfig)**

**Class:** com.cognizant.springlearn.config.SecurityConfig

java

package com.cognizant.springlearn.config;

import org.springframework.context.annotation.Configuration;

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

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

@Configuration

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests().antMatchers("/authenticate").permitAll()

.anyRequest().authenticated();

}

}

**Explanation**

**Reading the Authorization header**

* The Authorization header is received in Base64 (e.g., Basic dXNlcjpwd2Q=).
* It’s decoded to plain text user:pwd.

**Token generation**

* JWT is created using io.jsonwebtoken.Jwts.
* Contains:
  + Subject: username
  + Issued at: current time
  + Expiration: 10 minutes
* Signed with a secret key using HS256.

**Security configuration**

* CSRF disabled since we are only using basic token flow here.
* /authenticate is open for all to allow credential verification and token generation.

**SME Explanation Points**

**What happens in the controller?**

* Reads Authorization header.
* Decodes credentials.
* Validates user.
* Generates and returns JWT.

**How does Spring Security allow this request?**

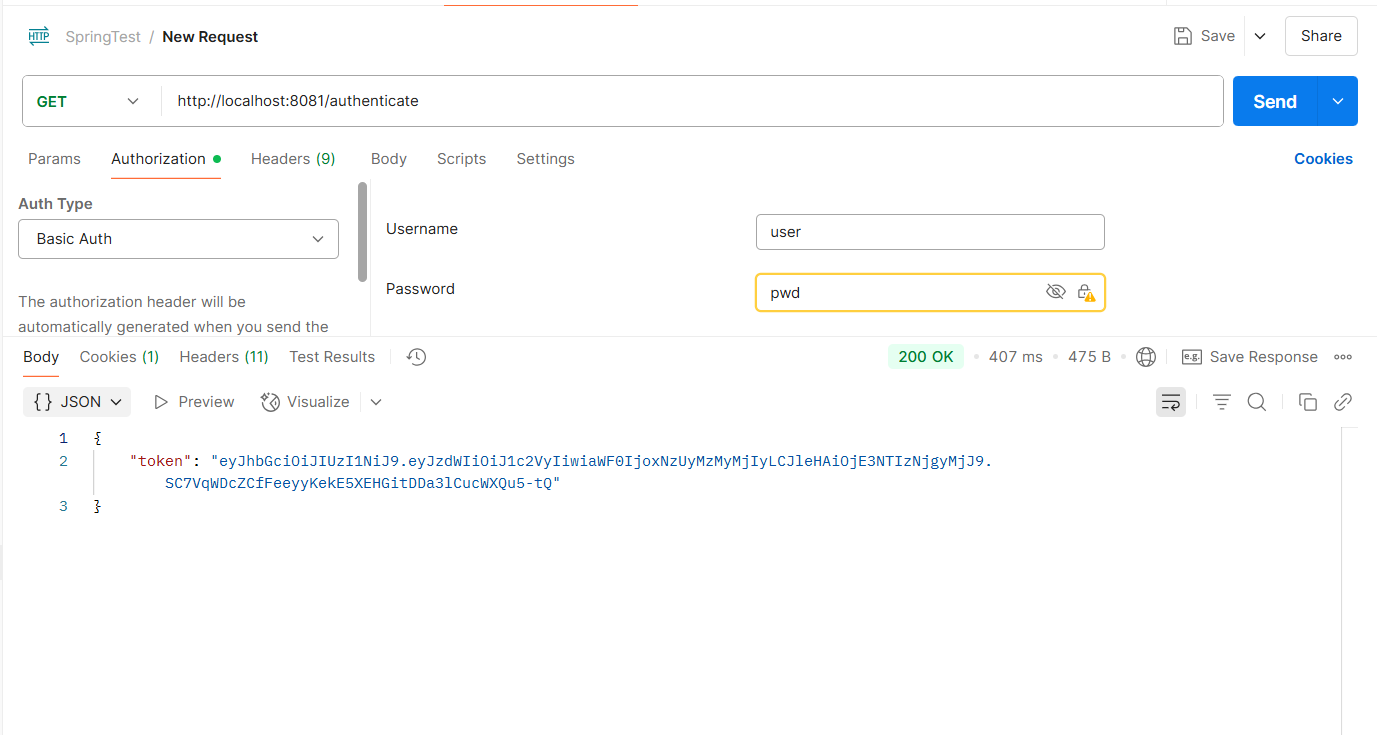
* Configured in SecurityConfig to **permit** /authenticate endpoint without authentication.

**Where is the token coming from?**

* Generated on-the-fly using **jjwt** library.
* Returned as JSON key token.

**Check headers in DevTools & Postman**

* In browser (DevTools → Network → Request Headers tab), you see:
  + Authorization: Basic dXNlcjpwd2Q=
* In Postman (Headers tab), you see:
  + Authorization: Basic dXNlcjpwd2Q=
  + Response content-type: application/json



**WEEK 4 ADDITIONAL HANDS ON**

**Employee List & Department RESTful Services**

**Problem Statement**

Previously, the Angular app used hardcoded employee data. We now need to:

* Store employee and department data in Spring XML configuration.
* Create REST APIs to serve this data.
* Update Angular to consume these APIs for listing and editing employees.

**Static Employee List Data in XML**

* We created an **employee.xml** configuration file.
* Added multiple departments (e.g., HR, Finance).
* Created 4 or more employee entries, referencing department and skills beans.
* Collected all employees in an **ArrayList**, defined as a bean.

**Employee DAO Layer**

* Created a DAO class to **read employee list** from employee.xml.
* Stored the list in a static variable.
* Provided a method to get all employees.

**Employee REST Service**

* Created a **Service class** to call DAO and return employee list.
* Marked service methods with @Transactional.
* Created a **Controller class** with a GET endpoint /employees.
* The controller calls service and returns the list as JSON

**Testing Employee API**

* URL: http://localhost:8083/employees
* Method: GET
* Tested using Postman.
* Confirmed JSON list of all employees is returned.

**Department REST Service**

* Created DAO, Service, and Controller classes for departments, similar to employee.
* Added endpoint /departments to get all departments.
* Data also sourced from employee.xml.

**Testing Department API**

* URL: http://localhost:8083/departments
* Method: GET
* Verified using Postman and browser network tab.

**SME Explanation Points**

* XML file acts as static data source instead of hardcoding.
* DAO layer handles reading from XML.
* Service layer provides a business layer and can handle transactions.
* Controller layer exposes REST endpoints, returning data as JSON.
* Angular frontend can consume these APIs dynamically instead of using static values.

