**WEEK 4-MANDATORY HANDS ON EXERCISES**

**Module 7 - Spring REST using Spring Boot 3**

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

**✅ Project Setup**

* **Group ID:** com.cognizant
* **Artifact ID:** spring-learn
* **Dependencies Selected:** Spring Boot DevTools, Spring Web
* **Build Tool:** Maven
* **IDE Used:** Eclipse

The project was generated from Spring Initializr, downloaded as a ZIP, extracted, and imported into Eclipse.

**⚙️ Project Structure**

**1️⃣ src/main/java**

This folder contains the **main application source code**. It includes:

* The com.cognizant.springlearn package (automatically created using the group and artifact ID).
* The SpringLearnApplication.java class which holds the main() method — this is the starting point of the Spring Boot application.

**2️⃣ src/main/resources**

This folder is used for **configuration files and static resources**, like:

* application.properties: Configuration file for Spring Boot (e.g., server port, database settings, etc.).
* Any static HTML, templates, or other resource files would also live here.

**3️⃣ src/test/java**

This folder contains **test classes** used for unit or integration testing.  
By default, a test class is generated for SpringLearnApplication to verify if the context loads successfully.

**💡 SpringLearnApplication.java**

@SpringBootApplication

public class SpringLearnApplication {

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

System.out.println("SpringLearnApplication started successfully!");

}

}

**🔥 Purpose of @SpringBootApplication**

This annotation is a **shortcut** for combining three annotations:

* @Configuration: Indicates that the class can be used by Spring IoC container as a source of bean definitions.
* @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and property settings.
* @ComponentScan: Enables component-scanning so that web controllers and other components are detected automatically.

In short — it bootstraps your whole application without writing boilerplate configurations. You get "magic", but good magic.

**🧾 pom.xml**

**Configuration Overview**

* Contains **project metadata** like groupId, artifactId, version.
* Declares **dependencies** such as:
  + spring-boot-starter-web: Adds Spring MVC and embedded Tomcat.
  + spring-boot-devtools: For live reload during development.
* Includes **build plugins** for Maven.

**Dependency Hierarchy**

In Eclipse:

* Right-click pom.xml → **Dependency Hierarchy** tab.
* It visually shows all dependencies and their transitive dependencies in a tree format.
* Helps in troubleshooting version conflicts and understanding which libraries are included.

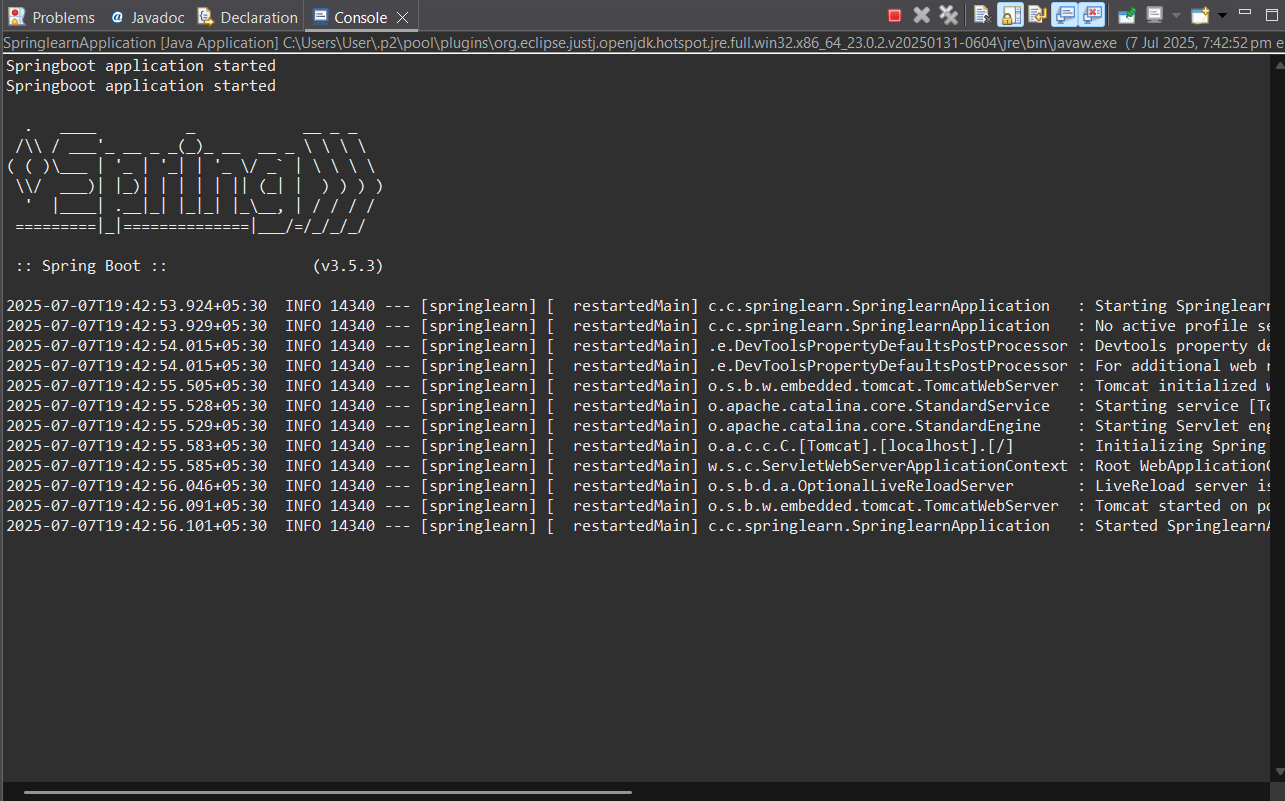
**⚡ Build & Run**

* Project built using:

mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456

* Imported as **Existing Maven Project** into Eclipse.
* Ran SpringLearnApplication.java as a Java Application.
* Verified logs to ensure that main() executed successfully and application started.

**✅ Summary**

****We have successfully created and run a Spring Boot Web project using Maven.  
The project structure follows Spring Boot conventions, the main class uses @SpringBootApplication to simplify configuration, and the pom.xml handles all dependency management using Maven.

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

**🛫 Context**

An airline website needs to support multiple countries, each identified by a two-character ISO code and a name.  
**For this exercise, we configure only one country: India (IN).**

**✅ Country Configured**

| **Code** | **Name** |
| --- | --- |
| IN | India |

**💡 Objective**

* Configure India as a Spring bean using XML (country.xml).
* Load this bean in the application and display its details.

**⚙️ Steps Implemented**

**1️⃣ Spring Configuration File (country.xml)**

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.springlearn.Country">

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

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

</bean>

</beans>

**2️⃣ Country Class**

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 + "]";

}

}

**3️⃣ displayCountry() Method**

public static void displayCountry() {

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

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

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

}

**4️⃣ Call in main()**

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

displayCountry();

}

**⚡ Logs**

When you run the main() method:

* You'll see debug logs showing constructor call, setter and getter messages, and final output for India.
* Confirms Spring container properly loaded the bean from XML.

**💬 SME Detailing**

**🟢 bean tag & attributes**

* <bean>: Declares a Spring-managed object.
* id: Unique identifier ("country").
* class: Java class name (com.cognizant.springlearn.Country).
* <property>: Sets values for the bean's fields.

**🟢 ApplicationContext & ClassPathXmlApplicationContext**

* ApplicationContext: Central interface for Spring container.
* ClassPathXmlApplicationContext: Loads XML from the classpath.

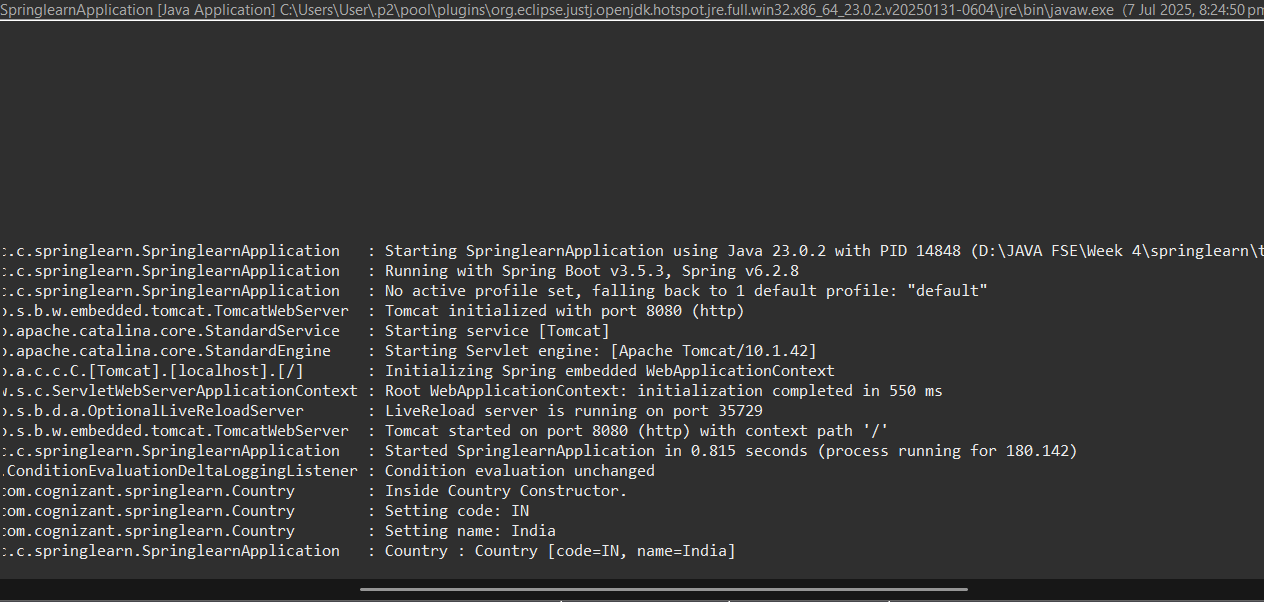
**🟢 context.getBean()**

* Parses XML.
* Creates bean object.
* Injects properties via setters.
* Returns fully initialized object.

**✅ Final Note**

Only India (IN) is configured and tested in this hands-on, as per instructions.  
The other country codes are provided for context and future extension if needed.





**Hello World RESTful Web Service**

**✅ Objective**

To create a REST service inside the **spring-learn** application that returns a simple text:

Hello World!!

when accessed via a **GET** request at /hello.

**⚙️ Implementation**

**1️⃣ Controller Class**

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

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.RestController;

@RestController

public class HelloController {

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

@GetMapping("/hello")

public String sayHello() {

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

String message = "Hello World!!";

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

return message;

}

}

**2️⃣ Endpoint Details**

* **Method:** GET
* **URL:** http://localhost:8083/hello
* **Response:** Hello World!!
* **Port:** 8083 (make sure you update application.properties to set server.port=8083)

**3️⃣ application.properties**

server.port=8083

**🌐 Testing the Service**

**Browser**

* Enter URL: http://localhost:8083/hello
* You’ll see:

Hello World!!

**Postman**

* Method: GET
* URL: http://localhost:8083/hello
* Click **Send**
* Response:

Hello World!!

**💬 SME Explanation Points**

**🟢 Network tab in Chrome Developer Tools**

* Open Chrome DevTools (F12 → Network tab).
* Refresh or send request to /hello.
* Click the request entry → Check **Headers** tab.
* You will see:
  + **Request headers**: e.g., Host, User-Agent, Accept, etc.
  + **Response headers**: e.g., Content-Type (text/plain;charset=UTF-8), Content-Length, Date, etc.

**🟢 Postman Headers tab**

* After sending request in Postman, click on **Headers** tab in the response section.
* You will see all **response headers** received from Spring Boot server, such as:
  + Content-Type: text/plain;charset=UTF-8
  + Transfer-Encoding: chunked
  + Date: current server timestamp
  + Connection: keep-alive

**⚡ Logs**

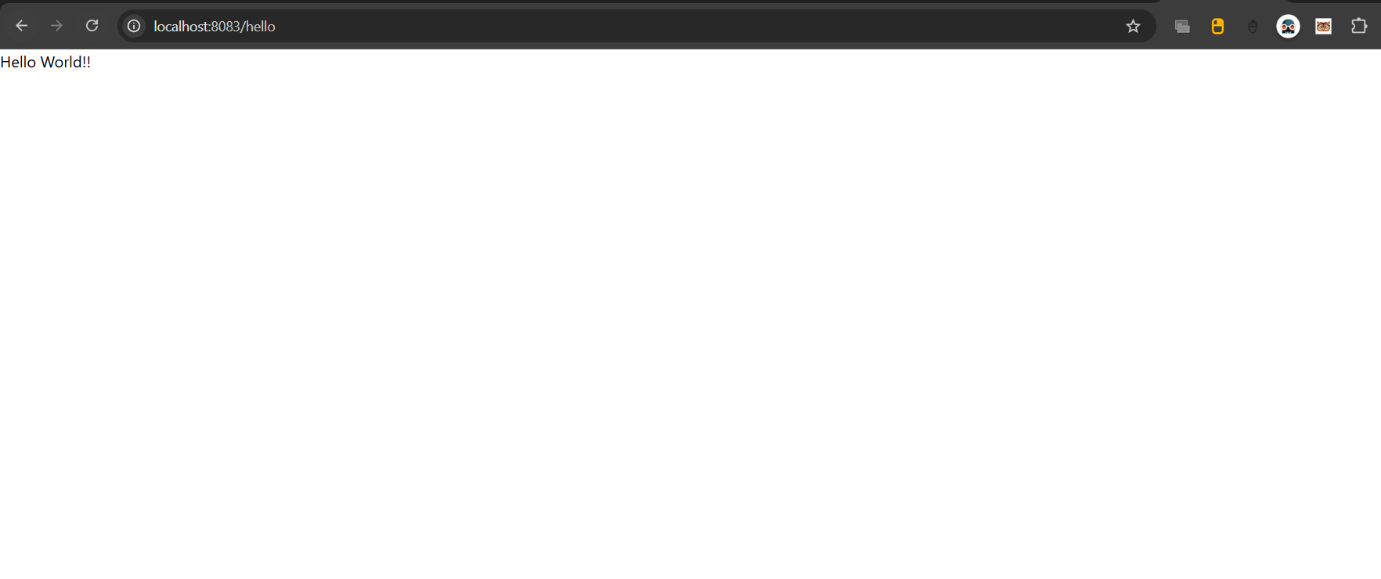
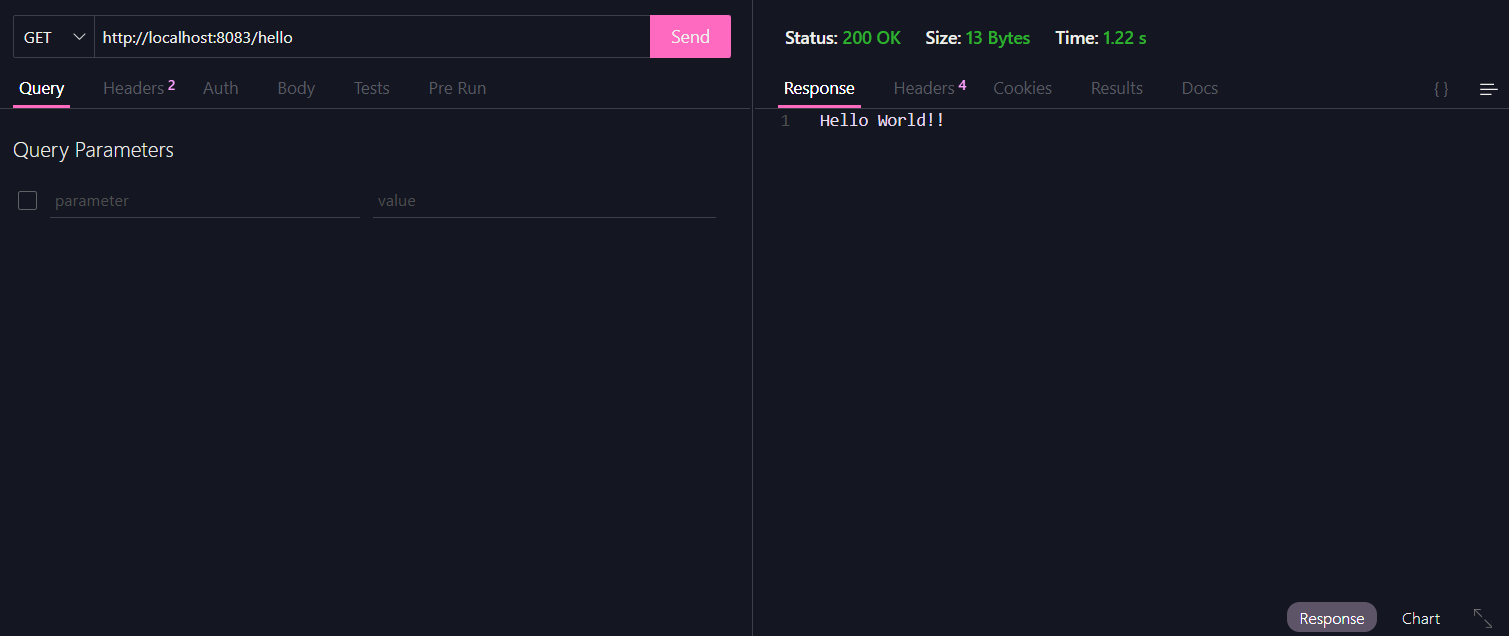
When /hello is accessed, logs show:

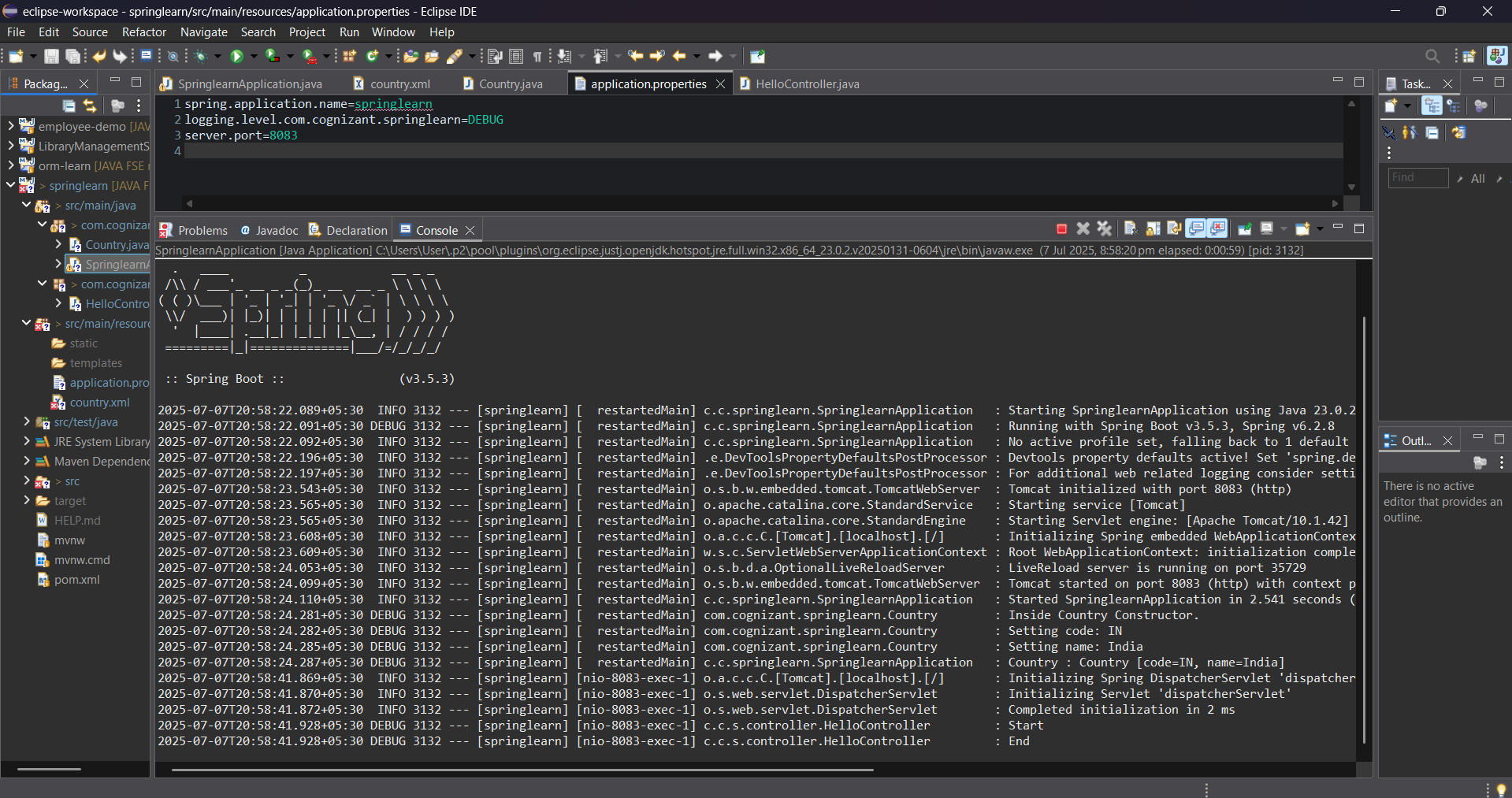
pgsql

DEBUG com.cognizant.springlearn.controller.HelloController - START: sayHello()

DEBUG com.cognizant.springlearn.controller.HelloController - END: sayHello()

**✅ Summary**

The /hello RESTful endpoint was successfully created using Spring Web, returning a simple "Hello World!!" text response. The endpoint works as expected in both browser and Postman, with clear logs to trace the execution.



**REST - Country Web Service**

**✅ Objective**

To create a REST service inside the **spring-learn** application that returns **India country details** as a JSON response.

**🌐 Endpoint Details**

* **URL:** http://localhost:8083/country
* **Method Annotation:** @RequestMapping
* **HTTP Method:** GET (default for @RequestMapping without specifying method)
* **Sample Response:**

json

{

"code": "IN",

"name": "India"

}

**⚙️ Implementation**

**1️⃣ Spring Configuration (country.xml)**

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.springlearn.Country">

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

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

</bean>

</beans>

**2️⃣ Country Class**

Same as before, with code, name, getters, setters, toString(), and debug logs.

**3️⃣ CountryController**

package com.cognizant.springlearn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.cognizant.springlearn.Country;

@RestController

public class CountryController {

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

@RequestMapping("/country")

public Country getCountryIndia() {

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

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

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

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

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

return country;

}

}

**⚡ How It Works**

**🟢 What happens in the controller method?**

* When /country URL is accessed, Spring routes the request to getCountryIndia().
* The method loads country.xml via ClassPathXmlApplicationContext.
* Retrieves the bean named "country" (India).
* Logs start, bean data, and end messages.
* Returns the Country object.

**🟢 How is the bean converted to JSON response?**

* The @RestController annotation internally combines @Controller + @ResponseBody.
* When Country object is returned, Spring Boot uses **Jackson library** (automatically included in Spring Web) to convert the Java object to JSON format in the HTTP response body.

**🌐 Testing the Service**

**Browser**

* URL: http://localhost:8083/country
* Shows JSON output directly.

**Postman**

* Method: GET
* URL: http://localhost:8083/country
* Response body: JSON with code and name.

**💬 SME Explanation Points**

**Network tab (Chrome Dev Tools)**

* Open DevTools → Network → Access /country.
* Click request → Headers tab:
  + **Request headers:** Host, User-Agent, Accept, etc.
  + **Response headers:** Content-Type (application/json), Date, Content-Length, etc.

**Postman Headers tab**

* Send GET request to /country.
* Click **Headers** in response:
  + Content-Type: application/json
  + Date, Server info, etc.

**✅ Logs**

Example logs when /country is called:

pgsql

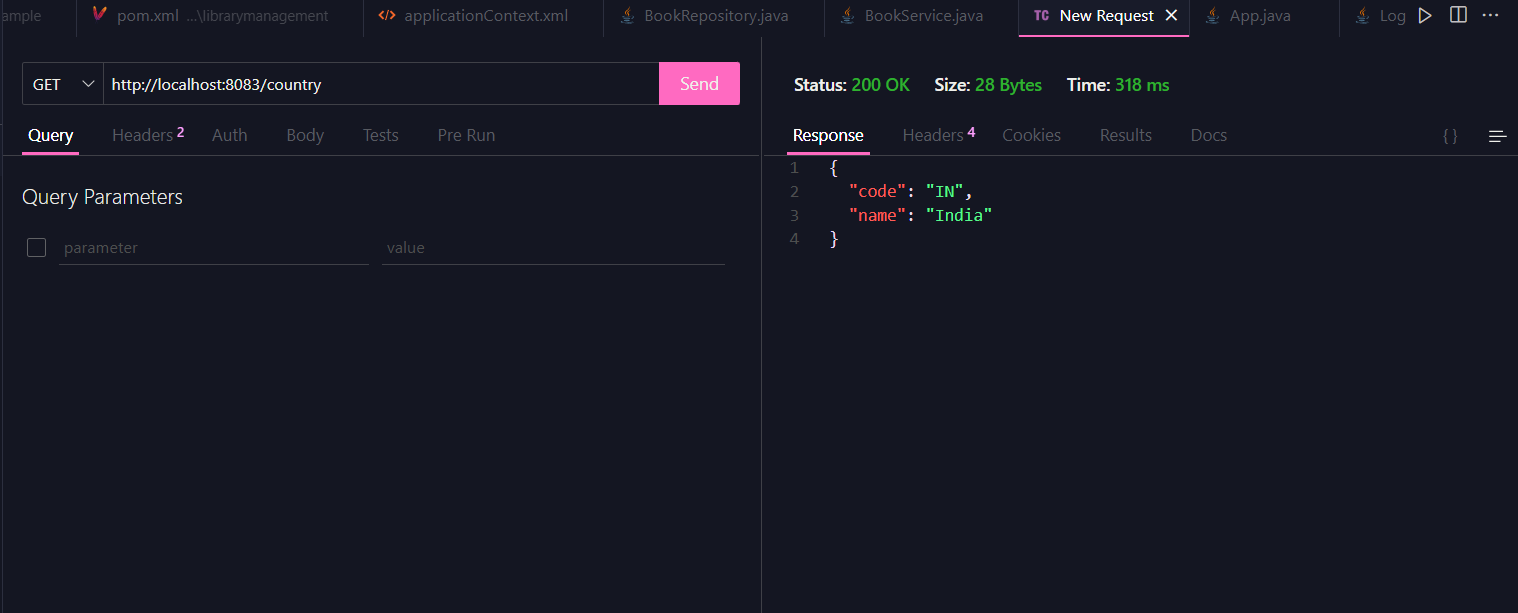
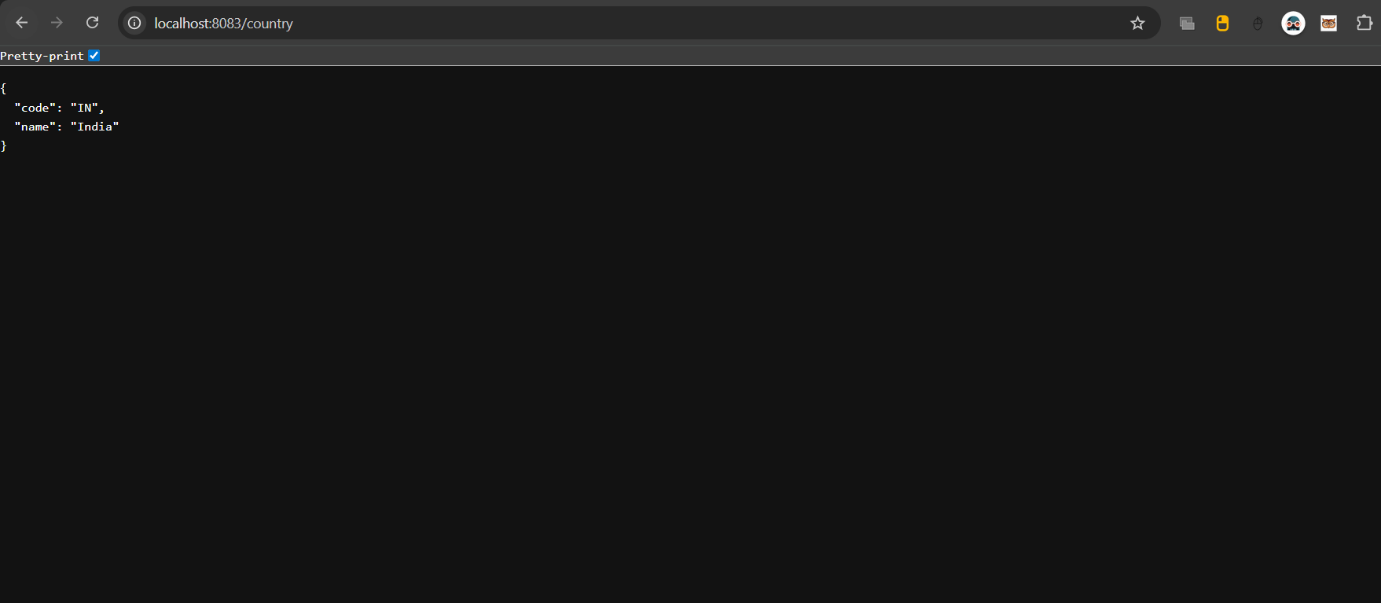
DEBUG com.cognizant.springlearn.controller.CountryController - START: getCountryIndia()

DEBUG com.cognizant.springlearn.controller.CountryController - Country: Country [code=IN, name=India]

DEBUG com.cognizant.springlearn.controller.CountryController - END: getCountryIndia()

**🟢 Final Summary**

The /country RESTful endpoint successfully returns India details as JSON using Spring XML configuration. The controller loads the bean, logs the flow, and Spring Boot automatically converts the bean to JSON. The endpoint can be tested in both browser and Postman.



**REST - Get Country Based on Country Code**

**✅ Objective**

Create a REST service that returns a specific country’s details based on its **country code**, ignoring case sensitivity.

**🌐 Endpoint Details**

* **URL:** http://localhost:8083/countries/{code}
* **HTTP Method:** GET
* **Path Variable:** code (e.g., in, US, etc.)
* **Sample Response:**

json

{

"code": "IN",

"name": "India"

}

**⚙️ Implementation**

**1️⃣ country.xml**

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="countryList" class="java.util.ArrayList">

<constructor-arg>

<list>

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

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

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

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

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

</bean>

</list>

</constructor-arg>

</bean>

</beans>

**2️⃣ CountryService**

**Class:** com.cognizant.springlearn.service.CountryService

package com.cognizant.springlearn.service;

import java.util.List;

import java.util.Optional;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.stereotype.Service;

import com.cognizant.springlearn.Country;

@Service

public class CountryService {

public Country getCountry(String code) {

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

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

Optional<Country> countryOptional = countryList.stream()

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

.findFirst();

if (countryOptional.isPresent()) {

return countryOptional.get();

} else {

throw new RuntimeException("Country not found with code: " + code);

}

}

}

**3️⃣ CountryController**

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

package com.cognizant.springlearn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

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

import com.cognizant.springlearn.Country;

import com.cognizant.springlearn.service.CountryService;

@RestController

public class CountryController {

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

@Autowired

private CountryService countryService;

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

public Country getCountry(@PathVariable String code) {

LOGGER.debug("START: getCountry() with code {}", code);

Country country = countryService.getCountry(code);

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

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

return country;

}

}

**💡 How It Works**

**🟢 Controller flow**

* URL /countries/{code} maps to getCountry() method.
* @PathVariable binds {code} from URL to method parameter.
* Method calls countryService.getCountry(code).
* Service method returns Country object, which is automatically converted to JSON.

**🟢 How is bean converted to JSON?**

* @RestController + Jackson library handle automatic conversion of Country Java object into JSON response.
* No extra code needed — Spring magic! ✨

**🧑‍💻 Testing**

**Browser**

* URL: http://localhost:8083/countries/in
* Displays JSON for India.

**Postman**

* Method: GET
* URL: http://localhost:8083/countries/in
* Response: JSON object for India.

**💬 SME Explanation Points**

**Network tab (Chrome Dev Tools)**

* Open DevTools → Network → Access /countries/in.
* Check **Headers** tab:
  + **Request headers**: Host, User-Agent, Accept.
  + **Response headers**: Content-Type (application/json), Content-Length, Date, etc.

**Postman Headers tab**

* After request, click on **Headers** in response section:
  + Content-Type: application/json
  + Date, Transfer-Encoding, Connection, etc.

**✅ Logs Example**

pgsql

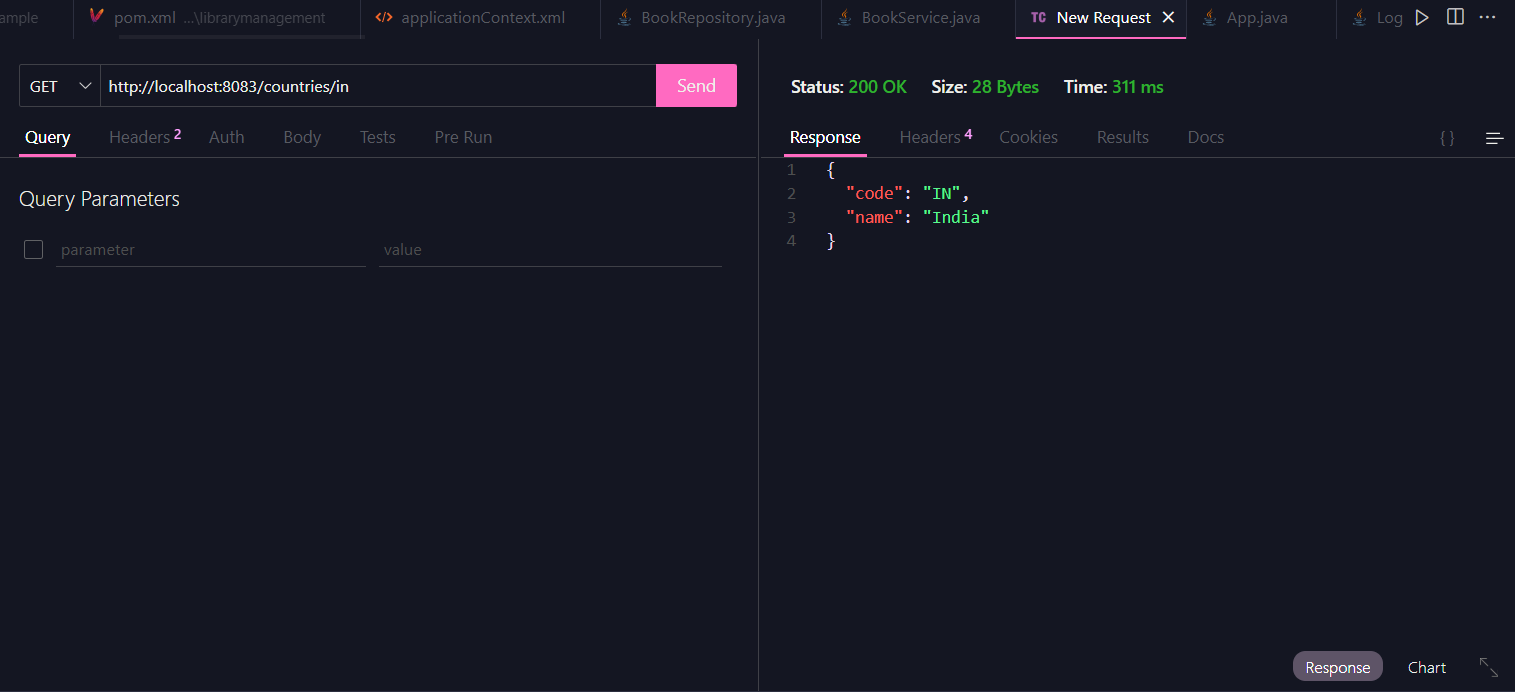
DEBUG com.cognizant.springlearn.controller.CountryController - START: getCountry() with code in

DEBUG com.cognizant.springlearn.controller.CountryController - Country: Country [code=IN, name=India]

DEBUG com.cognizant.springlearn.controller.CountryController - END: getCountry()

**🔥 Summary**

The /countries/{code} RESTful endpoint dynamically returns country details based on code, ignoring case. The controller delegates logic to service, which loads countries from Spring XML. Works seamlessly in both browser and Postman, with detailed logs and automatic JSON conversion.



**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**

curl -s -u user:pwd http://localhost:8090/authenticate

**💬 Example Response**

json

{

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

}

**🛠️ Implementation Steps**

**Step 1️⃣: 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 2️⃣: 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?**

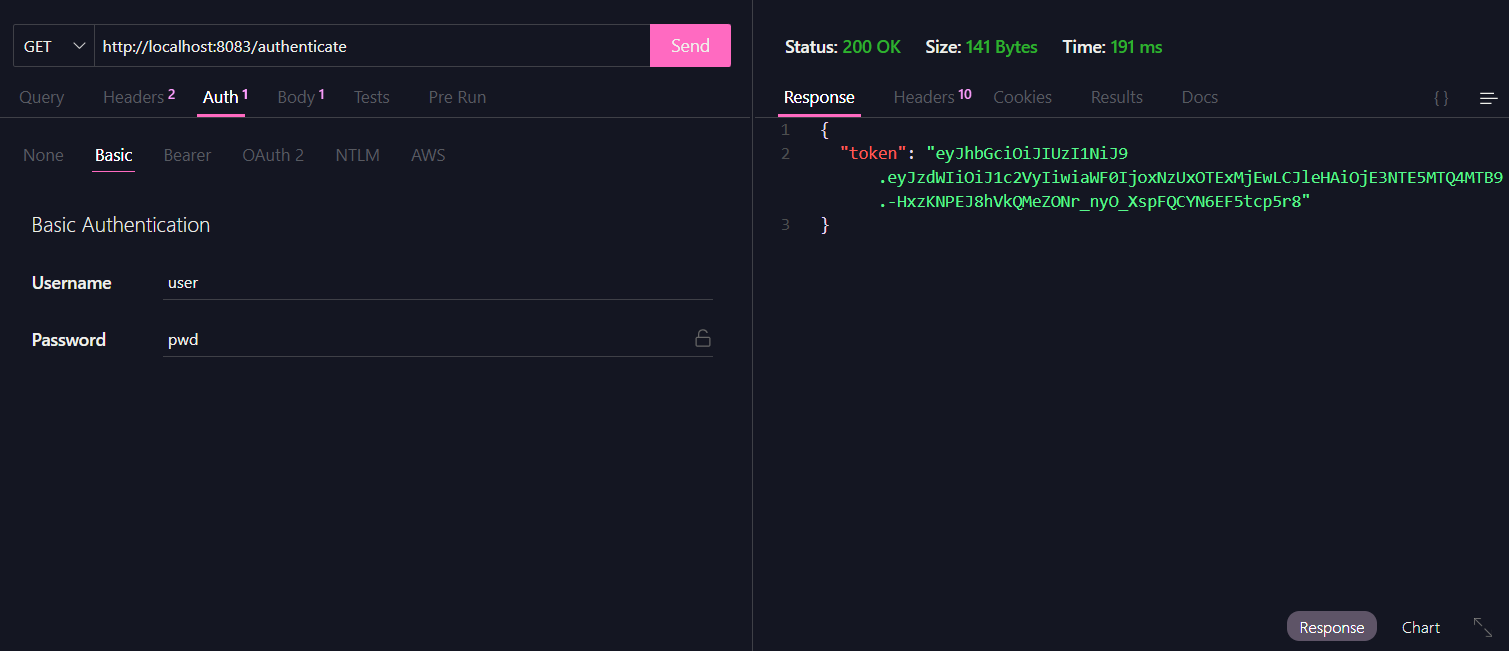
* Generated on-the-fly using **jjwt** library.
* Returned as JSON key token.

**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

**✅ Final Summary**

The /authenticate REST service validates credentials using Basic Auth and returns a signed JWT token as JSON. This is the first step to secure future requests in a JWT-based authentication system.



**WEEK 4-ADDITIONAL HANDS ON EXERCISES**

**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.

**🎯 Summary**

The static employee and department data are moved to Spring XML configuration and exposed via REST services.  
These services are tested and ready to integrate with Angular to show employee list and edit forms dynamically.

