WEEK - 4
Spring REST using Spring Boot

Create a Spring Web Project using Maven

Follow steps below to create a project:

- Go to https://start.spring.io/
- Change Group as “com.cognizant”
- Change Artifact Id as “spring-learn”
- Select Spring Boot DevTools and Spring Web
- Create and download the project as zip
- Extract the zip in root folder to Eclipse Workspace
- Build the project using ‘mvn clean package -Dhttp.proxyHost=proxy.cognizant.com -Dhttp.proxyPort=6050 -Dhttps.proxyHost=proxy.cognizant.com -Dhttps.proxyPort=6050 -Dhttp.proxyUser=123456’ command in command line
- Import the project in Eclipse "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"
- Include logs to verify if main() method of SpringLearnApplication.
- Run the SpringLearnApplication class.

SME to walk through the following aspects related to the project created:

- src/main/java - Folder with application code
- src/main/resources - Folder for application configuration
- src/test/java - Folder with code for testing the application
- SpringLearnApplication.java - Walkthrough the main() method.
- Purpose of @SpringBootApplication annotation
- pom.xml
    - Walkthrough all the configuration defined in XML file
    - Open 'Dependency Hierarchy' and show the dependency tree.

Here's a step-by-step guide and walk-through to create and understand a **Spring Web Project using Maven** with the specified configurations.

WEEK - 4

✅ PART 1: Create Spring Boot Project

1. **Go to Spring Initializer**
   - URL: https://start.spring.io
2. **Project Metadata:**
   - **Group**: `com.cognizant`
   - **Artifact**: `spring-learn`
   - **Name**: `spring-learn`
   - **Dependencies**:
     - Spring Boot DevTools
     - Spring Web
3. **Download the project**
   - Click **Generate**, and a ZIP will be downloaded.
4. **Extract ZIP**
   - Extract into Eclipse workspace (e.g., `C:\Users\<your-user>\eclipse-workspace\spring-learn`)

✅ PART 2: Build Project via Command Line

Navigate to project folder in command line and execute:

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

Replace `123456` with your employee ID (or actual proxy credentials if required).

WEEK - 4

✅ PART 3: Import Project in Eclipse

Steps:

- Open Eclipse
- Go to `File` → `Import`
- Select `Maven` → `Existing Maven Projects`
- Click **Browse**, select the extracted folder
- Click **Finish**

✅ PART 4: Run and Verify

1. Open `SpringLearnApplication.java` (under `src/main/java/com/cognizant/springlearn`)
2. Add a **log statement** in `main()` method to verify the application is starting:

```
@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);
        LOGGER.info("END");
    }
}
```

3. Right-click on `SpringLearnApplication.java` → `Run As` → `Java Application`

WEEK - 4

✅ PART 5: Project Structure Walkthrough

📁 `src/main/java`

- Contains application **Java source code**
- Default package: `com.cognizant.springlearn`

📁 `src/main/resources`

- Contains configuration files:
  - `application.properties` or `application.yml`
  - Static files, templates, or custom config

📁 `src/test/java`

- Contains **unit and integration tests**

✅ Understanding `SpringLearnApplication.java`

`@SpringBootApplication`

- Combination of:
  - `@Configuration`: Marks class as source of bean definitions
  - `@EnableAutoConfiguration`: Automatically configures Spring based on dependencies
  - `@ComponentScan`: Scans `com.cognizant.springlearn` and sub-packages for Spring components

`main()` method

- Entry point to run the Spring Boot application
- Uses `SpringApplication.run()` to bootstrap application

WEEK - 4

✅ Understanding `pom.xml`

Key Sections:

- `<groupId>`: `com.cognizant`
- `<artifactId>`: `spring-learn`
- `<dependencies>`:
  - Spring Boot Starter Web: Adds REST and web support
  - Spring Boot DevTools: Adds hot reload
- `<build>`: Contains plugin configuration (like `spring-boot-maven-plugin`)

Example snippet:

```
<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>
    <!-- Add other dependencies if needed -->
</dependencies>
```

✅ Dependency Hierarchy in Eclipse

- Right-click on project → `Maven` → `Show Dependency Hierarchy`
- Use it to:
  - Check **transitive dependencies**
  - Identify **version conflicts**
  - Understand what each starter brings in

WEEK - 4

✅ Summary

Component	Description
SpringLearnApplication.java	Entry point of the application
@SpringBootApplication	Enables auto-configuration, scanning, and config
`pom.xml`	Project dependencies and build config
src/main/java	Main source code
src/main/resources	Properties/configuration files
src/test/java	Test source code
Dependency Hierarchy	View and resolve Maven dependency tree

Spring Core – Load Country from Spring Configuration XML

An airlines website is going to support booking on four countries. There will be a drop down on the home page of this website to select the respective country. It is also important to store the two-character ISO code of each country.

Code	Name
US	United States
DE	Germany
IN	India
JP	Japan

Above data has to be stored in spring configuration file. Write a program to read this configuration file and display the details.
Steps to implement

- Pick any one of your choice country to configure in Spring XML configuration named country.xml.
- Create a bean tag in spring configuration for country and set the property and values

```
<bean id="country" class="com.cognizant.springlearn.Country">
    <property name="code" value="IN" />
    <property name="name" value="India" />
</bean>
```

WEEK - 4

- Create Country class with following aspects:
  - Instance variables for code and name
  - Implement empty parameter constructor with inclusion of debug log within the constructor with log message as “Inside Country Constructor.”
  - Generate getters and setters with inclusion of debug with relevant message within each setter and getter method.
  - Generate toString() method
- Create a method displayCountry() in SpringLearnApplication.java, which will read the country bean from spring configuration file and display the country details. ClassPathXmlApplicationContext, ApplicationContext and context.getBean(“beanId”, Country.class). Refer sample code for displayCountry() method below.

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

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

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

- Invoke displayCountry() method in main() method of SpringLearnApplication.java.
- Execute main() method and check the logs to find out which constructors and methods were invoked.

SME to provide more detailing about the following aspects:

- bean tag, id attribute, class attribute, property tag, name attribute, value attribute
- ApplicationContext, ClassPathXmlApplicationContext
- What exactly happens when context.getBean() is invoked

Here's a complete guide to implement the **Spring Core – Load Country from Spring Configuration XML** functionality as per your requirements:

✅ Step-by-Step Implementation

WEEK - 4

1. Create the `Country` class

Location: com.cognizant.springlearn.Country

```
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;
    }
```

WEEK - 4

```
@Override
    public String toString() {
        return "Country [code=" + code + ", name=" + name +
"]";
    }
}
```

2. Create the Spring XML configuration file

File: `country.xml`
Location: `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
       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>
```

3. Update `SpringLearnApplication.java`

Add the `displayCountry()` method:

```
package com.cognizant.springlearn;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCo
ntext;
```

WEEK - 4

```
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() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("country.xml");
        Country country = context.getBean("country",
Country.class);
        LOGGER.debug("Country : {}", country.toString());
    }
}
```

✅ Expected Console Log Output

When you run the application, the log will show messages like:

```
DEBUG com.cognizant.springlearn.Country - Inside Country
Constructor.
DEBUG com.cognizant.springlearn.Country - Inside setCode()
DEBUG com.cognizant.springlearn.Country - Inside setName()
DEBUG com.cognizant.springlearn.Country - Inside getCode()
DEBUG com.cognizant.springlearn.Country - Inside getName()
DEBUG com.cognizant.springlearn.SpringLearnApplication -
Country : Country [code=IN, name=India]
```

🧠 SME Notes and Concepts

📌 `bean` tag (Spring XML)

- **`<bean>`**: Used to define a Spring-managed object.
    - `id`: Unique identifier for the bean.
    - `class`: Fully qualified name of the Java class.

WEEK - 4

- `<property>`: Used to inject values into the bean's properties.
  - `name`: Matches the Java property (e.g., `setName()`)
  - `value`: The actual value to inject.

📌 `ApplicationContext` and `ClassPathXmlApplicationContext`

- **ApplicationContext**: Central interface for Spring's IoC container.
- **ClassPathXmlApplicationContext**: Loads configuration from XML in the classpath.
- Creates and manages beans defined in XML (`country.xml` in this case).

📌 What happens on `context.getBean("country")`

- Spring:
  1. Reads the XML configuration
  2. Instantiates the `Country` class
  3. Injects the `code` and `name` via `setCode()` and `setName()`
  4. Returns the fully initialized bean

At this point, constructor and setters will be invoked and debug logs generated accordingly.

✅ Summary Table

Concept	Description
`@SpringBootApplication`	Entry point annotation for Spring Boot
`country.xml`	Spring XML file to configure and inject dependencies
`ApplicationContext`	Spring container to manage beans
`ClassPathXmlApplicationContext`	Loads Spring XML files from classpath
`context.getBean()`	Instantiates and returns the bean configured in XML
`LOGGER.debug()`	Logs each method invocation for traceability

WEEK - 4

Hello World RESTful Web Service

Write a REST service in the spring learn application created earlier, that returns the text "Hello World!!" using Spring Web Framework. Refer details below:

Method: GET
URL: /hello
Controller: com.cognizant.spring-learn.controller.HelloController
Method Signature: public String sayHello()
Method Implementation: return hard coded string "Hello World!!"
Sample Request: http://localhost:8083/hello
Sample Response: Hello World!!

IMPORTANT NOTE: Don't forget to include start and end log in the sayHello() method.

Try the URL http://localhost:8083/hello in both chrome browser and postman.

SME to explain the following aspects:

- In network tab of developer tools show the HTTP header details received
- In postman click on "Headers" tab to view the HTTP header details received

Here's how to create a **Hello World RESTful Web Service** using Spring Boot Web in your `spring-learn` application, step-by-step.

✅ Step-by-Step Implementation

1. Ensure Spring Web is in your `pom.xml`

If you already added `Spring Web` via Spring Initializr, skip this. Otherwise, add:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

2. Create the `HelloController` class

Location: `com.cognizant.springlearn.controller.HelloController`

```
package com.cognizant.springlearn.controller;
```

WEEK - 4

```
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");
        String response = "Hello World!!";
        LOGGER.info("END");
        return response;
    }
}
```

3. Run Your Application

- Go to `SpringLearnApplication.java`
- Right-click → `Run As` → `Java Application`

Make sure your `application.properties` (in `src/main/resources`) sets the correct port:

```
server.port=8083
```

If the file doesn't exist, create it and add this line.

4. Test the Endpoint

✅ **Using Browser:**

- Open Chrome
- Visit: http://localhost:8083/hello
- You'll see:
  `Hello World!!`

WEEK - 4

✅ **Using Postman:**

- Method: `GET`
- URL: `http://localhost:8083/hello`
- Click **Send**
- You'll get the response:
  `Hello World!!`

✅ SME EXPLANATION

🔍 Network Tab in Browser Dev Tools

1. Press `F12` or `Ctrl+Shift+I` in Chrome → Go to **Network** tab
2. Reload `http://localhost:8083/hello`
3. Click on the `/hello` request
4. In **Headers** tab, you'll see:

📩 **Request Headers:**

- `Host`: localhost:8083
- `User-Agent`: Chrome user agent string
- `Accept`: text/html, application/xhtml+xml, etc.
- `Accept-Encoding`: gzip, deflate, br
- `Connection`: keep-alive

📨 **Response Headers:**

- `Content-Type`: `text/plain;charset=UTF-8`
- `Content-Length`: `14`
- `Date`: Current timestamp
- `Keep-Alive`: timeout value

WEEK - 4

🧪 Headers Tab in Postman

1. Send the request to `http://localhost:8083/hello`
2. Click on the "**Headers**" tab below the response

📩 Request Headers (automatically set by Postman):

- `User-Agent`: PostmanRuntime
- `Accept`: /
- `Host`: localhost:8083
- `Connection`: keep-alive

📩 Response Headers:

- `Content-Type`: text/plain;charset=UTF-8
- `Content-Length`: 14
- `Date`: Current server time

✅ Summary

Aspect	Description
URL	`http://localhost:8083/hello`
Method	GET
Controller	`HelloController`
Response	`"Hello World!!"`
Logging	`START` and `END` logged with SLF4J
Browser Dev Tools	Check headers under Network tab
Postman Headers Tab	View both request and response headers

WEEK - 4

REST - Country Web Service

Write a REST service that returns India country details in the earlier created spring learn application.

URL: /country
Controller: com.cognizant.spring-learn.controller.CountryController
Method Annotation: @RequestMapping
Method Name: getCountryIndia()
Method Implementation: Load India bean from spring xml configuration and return
Sample Request: http://localhost:8083/country
Sample Response:

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

SME to explain the following aspects:

- What happens in the controller method?
- How the bean is converted into JSON reponse?
- In network tab of developer tools show the HTTP header details received
- In postman click on "Headers" tab to view the HTTP header details received

Here's how to implement the **REST - Country Web Service** that returns **India's country details** using a Spring bean loaded from an XML file, and explain the internals as an SME would.

✅ Step-by-Step Implementation

🔷 1. XML Configuration

Ensure your `country.xml` (in `src/main/resources`) contains the India bean:

```
<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">
```

WEEK - 4

```
    <bean id="country"
class="com.cognizant.springlearn.Country">
        <property name="code" value="IN" />
        <property name="name" value="India" />
    </bean>

</beans>
```

🔷 2. Country POJO (`Country.java`)

Already created in previous step. Ensure it has:

- `code`, `name`
- `getters`, `setters`, `toString()`
- Debug logs in constructor and methods

🔷 3. Create `CountryController`

Location: `com.cognizant.springlearn.controller.CountryController`

```
package com.cognizant.springlearn.controller;

import com.cognizant.springlearn.Country;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCo
ntext;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RestController;

@RestController
public class CountryController {

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

WEEK - 4

```
    @RequestMapping("/country")
    public Country getCountryIndia() {
        LOGGER.info("START");

        ApplicationContext context = new
ClassPathXmlApplicationContext("country.xml");
        Country country = context.getBean("country",
Country.class);

        LOGGER.info("END");
        return country;
    }
}
```

🔷 4. Run the Application

Ensure your `application.properties` contains:

```
server.port=8083
```

Then run `SpringLearnApplication.java`.

🔍 Try It Out

📎 Request

- **URL:** http://localhost:8083/country
- **Method:** `GET`

📎 Sample Response

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

WEEK - 4

🧠 SME Explanation

🔶 What happens in the controller method?

1. `ApplicationContext` is initialized with `country.xml`.
2. Spring parses the XML and instantiates the `Country` bean.
3. The bean is returned from `getCountryIndia()`.
4. Spring Boot (via Jackson library) automatically **serializes the Java object to JSON**.

🔶 How is the bean converted into JSON?

- Spring Boot uses **Jackson** (a JSON library).
- `@RestController` implies `@ResponseBody`, which tells Spring to:
  - Call `getCountryIndia()`
  - Take the returned `Country` object
  - Use Jackson to convert it to JSON
  - Return the JSON as HTTP response

✅ No need to manually write JSON conversion logic.

🔶 HTTP Headers in Browser Dev Tools

1. Open Chrome
2. Visit `http://localhost:8083/country`
3. Press `F12` → Go to **Network** → Click `/country`

WEEK - 4

📩 **Response Headers:**

Header	Value
Content-Type	`application/json`
Content-Length	e.g., `33`
Date	Current timestamp

🔶 HTTP Headers in Postman

1. Open Postman
2. Send `GET` request to `http://localhost:8083/country`
3. Click on "**Headers**" tab

📩 **Response Headers in Postman:**

Header	Value
Content-Type	`application/json`
Content-Length	`33`
Server	`Apache Tomcat` or embedded

✅ Summary

Aspect	Description
Endpoint	`/country`
Controller	`CountryController`
Bean Source	Loaded from `country.xml`
Output Format	JSON (via Jackson auto-serialization)
Headers (browser/postman)	`Content-Type: application/json`
Framework Magic	`@RestController` + `Jackson` = JSON for POJO

WEEK - 4

REST - Get country based on country code

Write a REST service that returns a specific country based on country code. The country code should be case insensitive.

Controller: com.cognizant.spring-learn.controller.CountryController
Method Annotation: @GetMapping("/countries/{code}")
Method Name: getCountry(String code)
Method Implemetation: Invoke countryService.getCountry(code)
Service Method: com.cognizant.spring-learn.service.CountryService.getCountry(String code)

Service Method Implementation:

- Get the country code using @PathVariable
- Get country list from country.xml
- Iterate through the country list
- Make a case insensitive matching of country code and return the country.
- Lambda expression can also be used instead of iterating the country list

Sample Request: http://localhost:8083/country/in

Sample Response:

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

Here's how to implement the **REST API to get a country by country code** in the `spring-learn` project using Spring Boot and XML bean configuration:

✅ Step-by-Step Implementation

🔷 1. Update `country.xml` to include a list of countries

Location: `src/main/resources/country.xml`

WEEK - 4

```
<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">

    <!-- Individual Country Beans -->
    <bean id="country1"
class="com.cognizant.springlearn.Country">
        <property name="code" value="IN"/>
        <property name="name" value="India"/>
    </bean>

    <bean id="country2"
class="com.cognizant.springlearn.Country">
        <property name="code" value="US"/>
        <property name="name" value="United States"/>
    </bean>

    <bean id="country3"
class="com.cognizant.springlearn.Country">
        <property name="code" value="DE"/>
        <property name="name" value="Germany"/>
    </bean>

    <bean id="country4"
class="com.cognizant.springlearn.Country">
        <property name="code" value="JP"/>
        <property name="name" value="Japan"/>
    </bean>

    <!-- List of Countries -->
    <bean id="countryList" class="java.util.ArrayList">
        <constructor-arg>
            <list>
                <ref bean="country1"/>
                <ref bean="country2"/>
                <ref bean="country3"/>
                <ref bean="country4"/>
            </list>
        </constructor-arg>
    </bean>
```

```
</beans>
```

🔷 2. Create the `CountryService` class

Location: `com.cognizant.springlearn.service.CountryService`

```
package com.cognizant.springlearn.service;

import com.cognizant.springlearn.Country;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCo
ntext;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class CountryService {

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

    public Country getCountry(String code) {
        LOGGER.info("START - getCountry()");

        ApplicationContext context = new
ClassPathXmlApplicationContext("country.xml");
        List<Country> countryList = (List<Country>)
context.getBean("countryList");

        Country country = countryList.stream()
                .filter(c ->
c.getCode().equalsIgnoreCase(code))
                .findFirst()
                .orElse(null);

        LOGGER.info("END - getCountry()");
        return country;
    }
}
```

WEEK - 4

🔷 3. Update the `CountryController` class

Location: `com.cognizant.springlearn.controller.CountryController`

```
package com.cognizant.springlearn.controller;

import com.cognizant.springlearn.Country;
import com.cognizant.springlearn.service.CountryService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@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.info("START - getCountry()");
        Country country = countryService.getCountry(code);
        LOGGER.info("END - getCountry()");
        return country;
    }
}
```

🔷 4. Test the Endpoint

✅ **URL:**

```
http://localhost:8083/countries/in
```

✅ **Sample Response:**

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

WEEK - 4

🧠 SME Explanation

🔶 What happens in the controller method?

- The controller receives the request at `/countries/{code}`
- `@PathVariable` binds the `{code}` value to the method parameter
- It calls `CountryService.getCountry(code)`
- The service loads `countryList` from `country.xml`
- It searches for a matching country using `equalsIgnoreCase()`
- Returns the matching country object, which Spring converts to JSON

🔶 How is the Java object converted to JSON?

- Spring Boot includes **Jackson**, which auto-converts Java objects into JSON.
- Because the controller is annotated with `@RestController`, the return object is automatically serialized.

🔶 Network Tab (Dev Tools)

1. Open Chrome → F12 → Network tab
2. Visit `http://localhost:8083/countries/in`
3. Click the `/countries/in` request

HTTP Response Headers:

Header	Value
Content-Type	`application/json`
Content-Length	`33`
Date	Current time

WEEK - 4

🔶 Headers Tab in Postman

1. Send a `GET` request to `http://localhost:8083/countries/in`
2. Click on "**Headers**" tab

You'll see:

📩 **Response Headers:**

- `Content-Type`: `application/json`
- `Content-Length`: ~33
- `Date`: Server response time

✅ Summary

Component	Description
XML File	`country.xml` stores list of countries
Service Class	`CountryService.getCountry(code)`
REST Endpoint	`/countries/{code}`
Case Insensitive Match	Uses `equalsIgnoreCase()`
Output Format	JSON (auto-converted by Spring)
Dev Tools / Postman	View `Content-Type: application/json`

WEEK - 4

Create authentication service that returns JWT

As part of first step of JWT process, the user credentials needs to be sent to authentication service request that generates and returns the JWT.

Ideally when the below curl command is executed that calls the new authentication service, the token should be responded. Kindly note that the credentials are passed using -u option.

Request

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

Response

```
{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiaWF0IjoxNTcwMzc5NDc0LCJleHAiOjE1NzAzODA2NzR9.t3LRvlCV-hwKfoqZYlaVQqEUiBloWcWn0ft3tgv0dL0"}
```

This can be incorporated as three major steps:

- Create authentication controller and configure it in SecurityConfig
- Read Authorization header and decode the username and password
- Generate token based on the user retrieved in the previous step

Let incorporate the above as separate hands on exercises.

Great! You're about to build a **JWT authentication service** in Spring Boot. Let's begin with **Step 1: Create the authentication controller and configure it in the `SecurityConfig`**.

✅ Step 1: Authentication Controller + Security Configuration

🔷 1. Add Dependencies in `pom.xml`

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

WEEK - 4

🔷 2. Create the `AuthenticationController`

Location:
`com.cognizant.springlearn.controller.AuthenticationController`

```
package com.cognizant.springlearn.controller;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.HttpServletRequest;
import java.util.Base64;
import java.util.Date;

@RestController
public class AuthenticationController {

    private final String SECRET_KEY = "mySecretKey"; // move
to env or config in real apps

    @GetMapping("/authenticate")
    public ResponseEntity<?> authenticate(HttpServletRequest
request) {
        String authHeader =
request.getHeader("Authorization");

        if (authHeader == null || !
authHeader.startsWith("Basic ")) {
            return ResponseEntity.status(401).body("Missing
or invalid Authorization header");
        }

        String base64Credentials =
authHeader.substring("Basic ".length()).trim();
        byte[] decodedBytes =
Base64.getDecoder().decode(base64Credentials);
        String credentials = new String(decodedBytes);
        String[] values = credentials.split(":", 2);
```

```
        String username = values[0];
        String password = values[1];

        // Dummy check (replace with actual DB/userService
validation)
        if (!username.equals("user") || !
password.equals("pwd")) {
            return ResponseEntity.status(401).body("Invalid
credentials");
        }

        String token = Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new
Date(System.currentTimeMillis() + 10 * 60 * 1000)) // 10 mins
                .signWith(SignatureAlgorithm.HS256,
SECRET_KEY)
                .compact();

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

🔷 3. Create `SecurityConfig` to Permit `/authenticate`

Location: `com.cognizant.springlearn.config.SecurityConfig`

```
package com.cognizant.springlearn.config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.H
ttpSecurity;
import
org.springframework.security.config.annotation.web.configurat
ion.*;

@EnableWebSecurity
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {
```

WEEK - 4

```
@Override
    protected void configure(HttpSecurity http) throws
Exception {
        http
            .csrf().disable()  // Disable CSRF for testing
            .authorizeRequests()
            .antMatchers("/authenticate").permitAll()
            .anyRequest().authenticated()
            .and()
            .httpBasic(); // Allows basic auth header
    }
}
```

🔷 4. Run the Application and Test

✅ **Curl command:**

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

✅ **Expected Response:**

```
{
  "token":"<jwt-token>"
}
```

🧠 What's Happening?

Step	Description
1. `-u user:pwd`	Sends `Authorization: Basic <base64>` header
2. Controller	Decodes base64, extracts user/pass
3. Auth check	Validates hardcoded creds
4. Token creation	Uses JJWT to sign a token with subject = `user`
5. Response	Returns the token in JSON format

WEEK - 4